UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Validación, adaptación y reparación de planes en el contexto de smart cities

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Martínez Cuesta, Marcos

Tutor/a: Garrido Tejero, Antonio

CURSO ACADÉMICO: 2023/2024

Resum

Aquest projecte es centra en el desenvolupament d'un algorisme que s'integra amb l'eina de validació de plans VAL, dins del context dinàmic i desafiador de les ciutats intel·ligents. La tasca principal consisteix a dissenyar i posar en pràctica un algorisme capaç de reparar plans existents.

L'enfocament central implica utilitzar el PDDL (Planning Domain Definition Language) per definir un domini i un problema relacionat amb el context de les ciutats intel·ligents. A partir d'aquesta definició, l'algorisme proposat ofereix un conjunt d'accions que es poden afegir per a reparar el pla original. Aquestes accions no solament estan dissenyades per corregir el pla, sinó que també es presenten a l'usuari amb informació rellevant i contextualitzada, permetent-li triar l'acció més adequada per a cada situació específica.

L'objectiu és proporcionar a l'usuari un conjunt clar d'opcions, cadascuna recolzada per dades rellevants, facilitant així la presa de decisions informades. Una vegada triada una acció per a reparar el pla, l'algorisme genera informació detallada sobre el pla modificat i els nous objectius que aborda com a resultat de la reparació.

És important destacar que l'algorisme també contempla la possibilitat de no reparar el pla i, en el seu lloc, optar per una replanificació completa utilitzant un planificador. Això assegura la validesa del pla resultant i permet a l'usuari comparar entre la reparació parcial i una solució completament nova.

Paraules clau: Intel·ligència artificial; tècniques intel·ligents; planificació; ciutats intel·ligents; reparació

Resumen

Este proyecto se enfoca en el desarrollo de un algoritmo que se integra con la herramienta de validación de planes VAL, dentro del contexto dinámico y desafiante de las *smart cities*. La tarea principal consiste en diseñar y poner en práctica un algoritmo capaz de reparar planes existentes.

El enfoque central es la utilización de PDDL (Planning Domain Definition Language) para definir un dominio y un problema relacionados con el contexto de las *smart cities*. A partir de esta definición, el algoritmo propuesto ofrece un conjunto de acciones a las que se puede unir para reparar el plan original. Estas acciones no solo están diseñadas para corregir el plan, sino que también se presentan al usuario con información relevante y contextualizada, permitiéndole elegir la acción más adecuada para cada situación específica.

El objetivo es proporcionar al usuario un conjunto claro de opciones, cada una respaldada por datos relevantes, facilitando así la toma de decisiones informadas. Una vez que se elige una acción para reparar el plan, el algoritmo genera información detallada sobre el plan modificado y los nuevos objetivos que este aborda como resultado de la reparación.

Es importante destacar que el algoritmo también contempla la posibilidad de no reparar el plan y, en su lugar, optar por una replanificación completa utilizando un planificador. Esto asegura la validez del plan resultante y permite al

usuario comparar entre la reparación parcial y una solución completamente nueva.

Palabras clave: Inteligencia artificial; técnicas inteligentes; planificación; smart cities; reparación

Abstract

This project focuses on developing an algorithm that integrates with the VAL plan validation tool within the dynamic and challenging context of smart cities. The main task is to design and implement an algorithm capable of repairing existing plans.

The central approach involves using PDDL (Planning Domain Definition Language) to define a domain and a problem related to the context of smart cities. Based on this definition, the proposed algorithm offers a set of actions that can be appended to repair the original plan. These actions are not only designed to correct the plan but also presented to the user with relevant and contextualized information, enabling them to choose the most suitable action for each specific situation.

The goal is to provide the user with a clear set of options, each supported by relevant data, thereby facilitating informed decision-making. Once an action is chosen to repair the plan, the algorithm generates detailed information about the modified plan and the new objectives it addresses as a result of the repair.

It is important to note that the algorithm also considers the possibility of not repairing the plan and instead opting for a complete replanning using a planner. This ensures the validity of the resulting plan and allows the user to compare between partial repair and an entirely new solution.

Key words: Artificial intelligence; intelligent techniques; planning; smart cities; repair

Índice general

Índice de figuras

Índice de algoritmos

CAPÍTULO 1
Introducción

El entorno urbano contemporáneo ha sido moldeado por un continuo proceso de transformación y adaptación a lo largo de la historia. Desde la Revolución Industrial hasta la era de la digitalización, las ciudades han sido el epicentro de la innovación y el cambio social. Sin embargo, es en el siglo XXI donde el concepto de *smart cities* ha ganado relevancia como respuesta a los desafíos urbanos modernos.

Las *smart cities* representan un paradigma emergente que busca no solo optimizar la eficiencia de las infraestructuras urbanas, sino también mejorar la calidad de vida de sus habitantes mediante la integración de tecnologías de vanguardia. Este enfoque holístico abarca desde la gestión inteligente de la energía, el uso eficiente de los recursos y la conectividad digital hasta la promoción de la participación ciudadana.

En el ámbito de la movilidad urbana, la integración de taxis y autobuses autónomos representa una innovación de gran alcance. Más allá de ser simples vehículos automatizados, estos medios de transporte autónomos están transformando la manera en que concebimos el desplazamiento urbano. Desde la optimización de rutas hasta la reducción de emisiones y la mejora de la seguridad vial, estos vehículos prometen cambios significativos en la experiencia de movilidad de los ciudadanos.

Además de la movilidad, las *smart cities* se centran en la optimización de la gestión de recursos. La monitorización en tiempo real de la demanda energética y el uso eficiente de la misma se convierten en pilares fundamentales para reducir el impacto ambiental y mejorar la sostenibilidad. Sistemas inteligentes de distribución de energía, junto con fuentes renovables, promueven una mayor autonomía energética en la ciudad.

La infraestructura de una *smart city* no solo se basa en tecnologías punta, sino también en la inclusión y participación ciudadana. Plataformas digitales que facilitan la interacción entre los residentes y las autoridades municipales fomentan una toma de decisiones más colaborativa y adaptada a las necesidades reales de la comunidad.

Además, se promueve una planificación urbana más inteligente que abarca desde la distribución equitativa de servicios hasta la gestión eficiente de residuos. La implementación de sensores y sistemas de recogida inteligente optimiza la recolección y el tratamiento de residuos, minimizando así el impacto ambiental.

Los edificios también se integran en esta visión de ciudades inteligentes. El uso de materiales sostenibles, junto con la implementación de sistemas de automatización para la gestión de energía y agua, contribuye a reducir el consumo y los costos a largo plazo.

En resumen, las *smart cities* son ecosistemas urbanos donde la tecnología y la innovación se combinan para mejorar la calidad de vida de los habitantes, promoviendo la sostenibilidad, la eficiencia y la participación ciudadana en todos los aspectos de la vida urbana.

1.1 Objetivos

El objetivo principal del trabajo será el de validar, adaptar y reparar planes, todo ello dentro del contexto relacionado con las *smart cities* y dando la posibilidad al usuario de elegir cómo reparar su plan. Para poder alcanzar este objetivo principal serán necesarios cumplir unos subobjetivos que se explican a continuación:

- **Formalizar un dominio y un problema en PDDL (Planning Domain Definition Language), centrado en los taxis y autobuses autónomos.** Esto implica la creación de un conjunto de reglas, acciones, estados y objetivos en PDDL que modelen el comportamiento de los taxis y autobuses autónomos dentro de un entorno específico, considerando variables como ubicaciones, destinos, pasajeros,

- **Integración entre la validación del plan y la capacidad de adaptación del plan generado.** Posterior a la generación de un plan, se requiere una fase de validación para determinar su idoneidad y ejecución efectiva. En caso de discrepancias o contingencias que afecten la ejecución, se implementará un proceso de adaptación o reparación del plan, considerando cambios o imprevistos en el entorno.

- **Aplicación de heurísticas para la adaptación ante la ruptura del plan.** Las heurísticas proporcionan una evaluación relativa entre las acciones del plan, ofreciendo una medida para diferenciar cuáles podrían ser más efectivas al ser combinadas y cuáles podrían no ser tan ventajosas.

- **Facilitación de opciones al usuario para la selección de acciones o la generación de nuevos planes.** Involucrar al usuario en el proceso de toma de decisiones. Ofrecer una gama de opciones claras y comprensibles para modificar acciones o generar nuevos planes proporciona al usuario un mayor control y comprensión del proceso de resolución de problemas.

1.2 Estructura de la memoria

La memoria se divide en cinco capítulos:

- **Marco teorico:** se abordarán las bases fundamentales necesarias para comprender adecuadamente el proyecto, incluyendo aspectos como las *smart cities* y la planificación, así como todos los elementos que intervienen en estos contextos.
- **Diseño de la solución:** En esta sección se llevará a cabo la modelización del dominio y problema empleados en el proyecto, detallando el proceso de validación, reparación y adaptación de planes de manera general, acompañado de un ejemplo simplificado para una comprensión más clara.
- **Implementación:** Esta sección se centrará en las tecnologías y herramientas clave empleadas en el proyecto, así como en los algoritmos diseñados para la validación precisa y la subsiguiente corrección de planes, respaldado por un ejemplo de ejecución para ilustrar el proceso.
- **Conclusión:** Se expone el resultado obtenido a partir del trabajo realizado, su relación con los estudios cursados y el posible trabajo futuro a desarrollar.

CAPÍTULO 2
Marco Teórico

Este capítulo tiene como objetivo asentar las bases teóricas necesarias para la correcta comprensión de los principales elementos que comprenden el proyecto. Para ello, empezaremos definiendo en qué consiste una *Smart City*, centrándonos en la parte en la que nos enfocaremos en el proyecto, seguidamente definiremos que es la planificación, comentando además la validación y la reparación de los planes obtenidos junto con la explicación de un grafo de planificación. Finalmente, explicaremos el lenguaje PDDL y sus principales elementos junto con un pequeño ejemplo para su correcta comprensión.

2.1 Definición de *Smart City*

El concepto de *'smart city'* o ciudad inteligente ha cobrado gran relevancia en la actualidad debido a la transformación significativa que experimentan las ciudades en todo el mundo. Esta transformación está impulsada por la tecnología y la necesidad de resolver desafíos urbanos. Las *smart cities* se perciben como la solución para diversos retos humanos. Cada persona destaca aspectos diferentes: algunos las ven como centros urbanos sostenibles con un enfoque en energías renovables (Motyka et al., 2019), mientras que otros resaltan su aspecto digital. También hay quienes piensan que deberían enfocarse en el desarrollo humano mediante la educación (Hambleton, 2014), entre otras perspectivas.

La realidad es que las ciudades inteligentes aún carecen de una definición consensuada, ya que siguen en proceso de construcción. Sin embargo, existe cierto consenso en que estas ciudades surgirán de la combinación entre la llamada 'cuarta revolución industrial' y el crecimiento demográfico previsto por las Naciones Unidas para los asentamientos urbanos (Gil-Casares y Ortiz, 2019; UN, 2019b).[3]

Una *smart city* es un entorno urbano que utiliza tecnologías de la información y las comunicaciones (TIC) para mejorar la calidad de vida de sus ciudadanos, optimizar la gestión de recursos y fomentar la sostenibilidad ambiental. Estas ciudades integran diversas tecnologías y sistemas para mejorar aspectos cruciales de la vida urbana, como la infraestructura, la movilidad, la seguridad y la gestión de residuos. La Figura 2.1 muestra algunos de los principales objetivos de las *smart cities*. Este trabajo se enfocará específicamente en la reducción de la

contaminación en estas ciudades, con un enfoque centrado en la movilidad mediante taxis y autobuses autónomos.

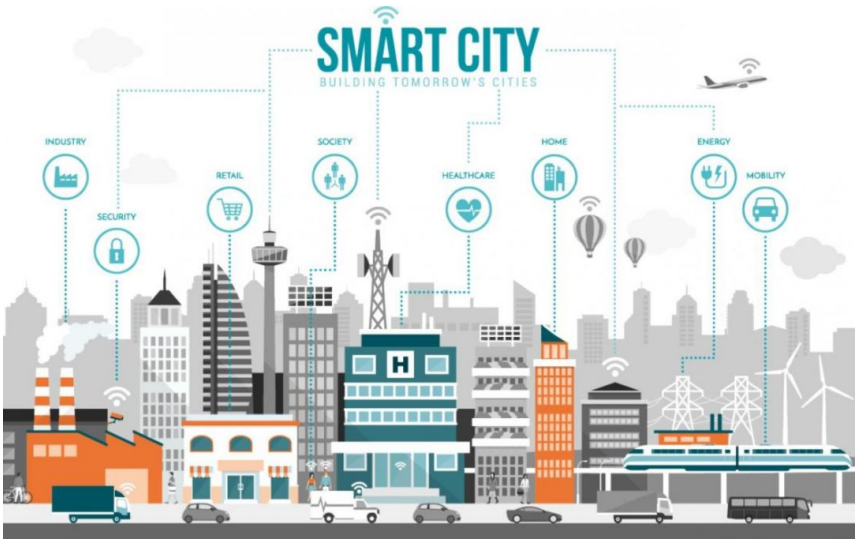

Figura 2.1: Diversos sectores en los que se enfoca una smart city.
Imagen recuperada de: https://protecciondatoscertificado.es/smart-cities/

2.1.1. Principales objetivos de una smart city

En la búsqueda por transformar entornos urbanos convencionales en espacios más eficientes, sostenibles y habitables, las *smart cities* han surgido como el paradigma de la innovación. Su objetivo principal radica en integrar tecnologías avanzadas para abordar desafíos multidisciplinarios, mejorando así la calidad de vida de sus habitantes. Desde economía y medio ambiente hasta movilidad y gobernanza, estas ciudades inteligentes persiguen una evolución holística que busca beneficiar tanto a la comunidad como al entorno que la rodea, así pues los objetivos son los siguientes:

- **Economía:** Crear un sistema económico online que ahorre tiempo y posibilite las transacciones de una manera sencilla para los ciudadanos.
- **Medio-ambiente:** El mayor aprovechamiento de recursos, así como la integración de la ciudad con el menor impacto sobre el entorno que la rodea debe ser una de las razones de mayor peso.
- **Gobernanza:** La digitalización de las administraciones públicas que faciliten las gestiones es un claro ejemplo a seguir, así como la transparencia en la gestión para con los ciudadanos.
- **Estilo de vida:** Edificios más eficientes, seguros y cómodos para sus habitantes.
- **Movilidad:** Sistemas para conocer el estado del tráfico, gestionarlo y optimizarlo es fundamental para la mejora organizativa y mejora de la calidad de vida de los ciudadanos. Uso de energías renovables y menores tiempos

de desplazamiento con una movilidad inteligente repercuten en más tiempo para las personas. Dentro de movilidad es donde encontramos:

1. **Taxis Autónomos:** Los taxis autónomos son vehículos que operan sin intervención humana. Estos vehículos están equipados con sensores, sistemas de navegación avanzados y tecnología de inteligencia artificial que les permite tomar decisiones en tiempo real y navegar de manera segura por las calles de la ciudad.
2. **Autobuses Autónomos:** Los autobuses autónomos son una extensión del concepto de transporte público en las *smart cities*. Estos vehículos ofrecen ventajas similares a los taxis autónomos y, además, pueden transportar a un mayor número de pasajeros, lo que contribuye a la reducción del tráfico y la congestión en las ciudades.

- **Personas:** Podemos decir que, junto con el medio-ambiente, forman el otro motivo de peso por el que las ciudades inteligentes se desarrollan. Ofrecer soluciones tomando como base las ideas y necesidades de las personas. [9]

2.2 Planificación

El propósito de la planificación es sintetizar un conjunto organizado de acciones para llevar a cabo alguna actividad. Por ejemplo, esto se puede lograr mediante un procedimiento de anticipación que combina pasos de predicción en una búsqueda a través de conjuntos alternativos de acciones para encontrar un conjunto que conduzca a un estado objetivo deseado.[1]
Es un proceso de razonamiento inteligente para seleccionar y organizar un conjunto de acciones que permita la consecución de unos objetivos en base a su causalidad como se muestra en la Figura 2.2. Y también es un proceso deliberativo en el que tratamos de anticiparnos a los efectos de las acciones, por lo que es previo a la ejecución (actuación) de las acciones.

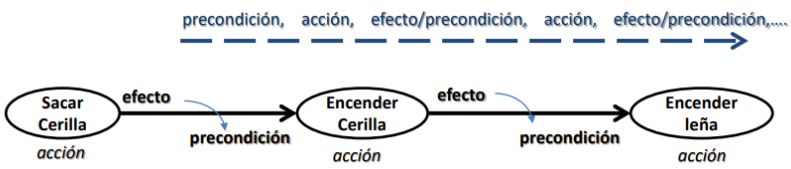

Figura 2.2: Ejemplo de causalidad en Planificación.
Imagen recuperada de: material docente y transparencias de poliformat de la asignatura Técnicas, Entornos y Aplicaciones de Inteligencia Artificial

Debido a la gran cantidad de decisiones que se tiene que tomar a la hora de la planificación, se opta por el uso de heurísticas. Estas estrategias se apoyan en métricas específicas que varían según la heurística empleada. Por ejemplo, algunas se basan en la maximización del rendimiento, otras en la minimización de costos o tiempos, y otras consideran la satisfacción de ciertos criterios predefinidos. Permiten abordar situaciones donde la búsqueda exhaustiva de la mejor opción resulta poco viable debido a restricciones de tiempo o recursos.

2.2.1. Heurísticas de planificación

Al emplear heurísticas, se busca alcanzar soluciones aceptables o satisfactorias en contextos donde la búsqueda exhaustiva de la mejor opción resulta costosa o poco práctica. Estas heurísticas se basan en principios de simplificación, asociación de ideas, reglas prácticas y experiencia previa para guiar el proceso de planificación hacia resultados viables.

2.2.1.1. Heurística por relajación

La heurística de relajación se centra en simplificar ciertos aspectos de un dominio y problema de planificación. Estos aspectos pueden variar según la relevancia para el problema en cuestión. La técnica más común implica relajar los efectos de eliminación (*delete effects*) de las acciones, lo que significa que una vez se alcanza un estado, este no se revierte. Sin embargo, existen otras áreas que pueden relajarse, como los valores numéricos, la gestión de recursos o las duraciones de las acciones.

La aplicación más sencilla de esta heurística es mediante el uso de grafos de planificación. Estos grafos representan una forma eficiente de explorar las posibles acciones desde un estado inicial hacia estados sucesores. Imagina un árbol que muestra todas las acciones posibles desde el estado inicial hacia sus sucesores y así sucesivamente. Si logramos indexar este árbol de manera adecuada, podríamos responder a la pregunta de planificación "¿podemos alcanzar el estado objetivo G desde el estado inicial S0?"de forma instantánea. No obstante, debido al tamaño exponencial del árbol, esta aproximación resulta poco práctica. Aquí es donde entra en juego el grafo de planificación, una representación polinómica aproximada de este árbol que se construye rápidamente.

El grafo de planificación no puede asegurar si es posible llegar definitivamente desde S0 a G, pero puede estimar la cantidad de pasos necesarios para alcanzar G. Esta estimación siempre es precisa al afirmar que el objetivo no es alcanzable y nunca sobreestima el número de pasos, lo que la convierte en una heurística admisible [5].

Este grafo se compone de dos tipos de niveles: el nivel de proposición, que contiene los nodos de estado, y el nivel de acción, que alberga los nodos representando las acciones ejecutables desde las proposiciones del nivel anterior. Además, hay tres tipos de aristas entre estos niveles: aristas-pre que van del nivel de proposición al nivel de acción, y aristas-add y aristas-del que van del nivel de acción al nivel de proposición. Este intercambio entre niveles genera el grafo completo. Una representación gráfica de este tipo de grafo se muestra en la Figura 2.3.

Esta estructura de grafo de planificación ofrece una forma eficaz de estimar la viabilidad y la cantidad de pasos necesarios para alcanzar un objetivo desde un estado inicial.

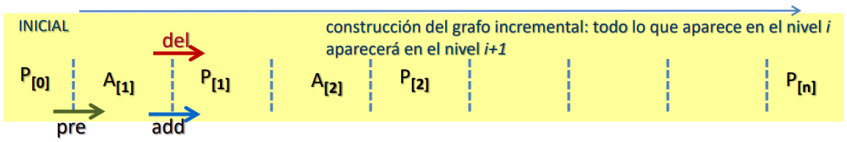

Figura 2.3: Esquema de un grafo de planificación.
Imagen recuperada de: material docente y transparencias de poliformat de la asignatura Técnicas, Entornos y Aplicaciones de Inteligencia Artificial

2.2.1.2. Heurística por abstracción

La abstracción se clasifica principalmente en dos categorías: la abstracción de estado, que implica representar únicamente las propiedades más relevantes del mundo, y la abstracción de acción, que implica la creación de un conjunto significativo de comportamientos a largo plazo disponibles. En ambos casos, la abstracción se entiende como 'el proceso de transformar la representación de un problema en una nueva representación'. La efectividad de los diferentes grados y tipos de abstracción varía según el problema en cuestión.

La abstracción de estado determina qué cambios en el entorno se consideran sustanciales, mientras que la abstracción de acción define las transiciones entre nodos o estados. En contraste, las acciones 'primitivas' representan las opciones más básicas disponibles. La abstracción de acción se ha referido con diferentes términos, como habilidades, abstracción temporal o macro-acciones. Sin embargo, ambos tipos de abstracción están estrechamente interconectados.

Existe una estrecha relación entre los dos tipos de abstracción: desde ciertas perspectivas, una acción abstracta simplemente representa un comportamiento que conduce a un agente de un estado abstracto a otro.[17]

La abstracción determina qué estados se considerarán idénticos y cuáles no. La estimación se calcula para el nuevo estado reducido y compacto, basado en variables y sus dominios asociados.

Un posible ejemplo de abstracción es el uso de variables de estado de forma no proposicional, es decir, se hace referencia a elementos que no se representan directamente como proposiciones lógicas o en un formato de lógica proposicional, por lo que estas variables pueden tomar formas más complejas o estructuradas que no se limitan a ser simplemente proposiciones verdaderas o falsas, con esto ya no es necesario los efectos *add* y *delete* de las acciones y nos permite la creación de otras estructuras como el grafo de transiciones, se puede observar un ejemplo en la Figura 2.4.[5]

Una vez obtenido este grafo se puede empezar con su abstracción, ya que hay muchas posibles abstracciones, que pueden conducir a mejores o peores costes dependiendo de la que se use, en la Figura 2.5 se pueden observar tres de ellas, cada una con una estimación distinta.

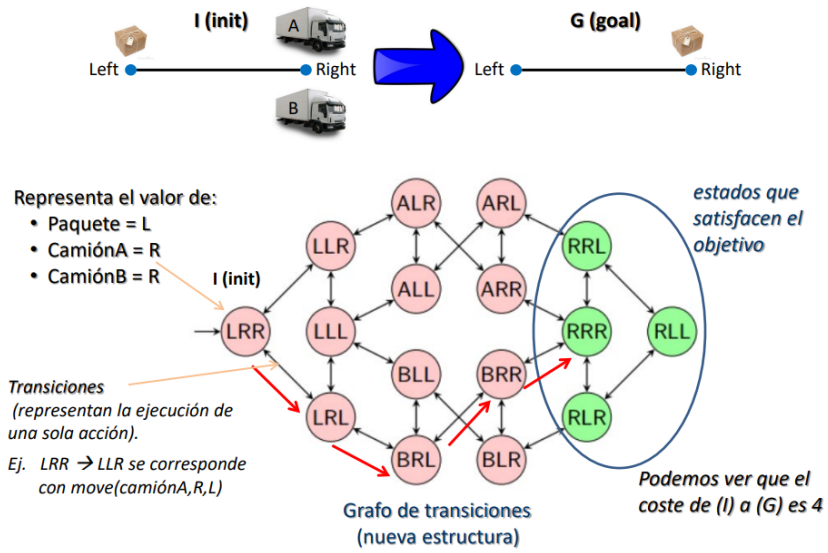

Figura 2.4: Ejemplo de un grafo de transición.
Imagen recuperada de: material docente y transparencias de poliformat de la asignatura Técnicas, Entornos y Aplicaciones de Inteligencia Artificial

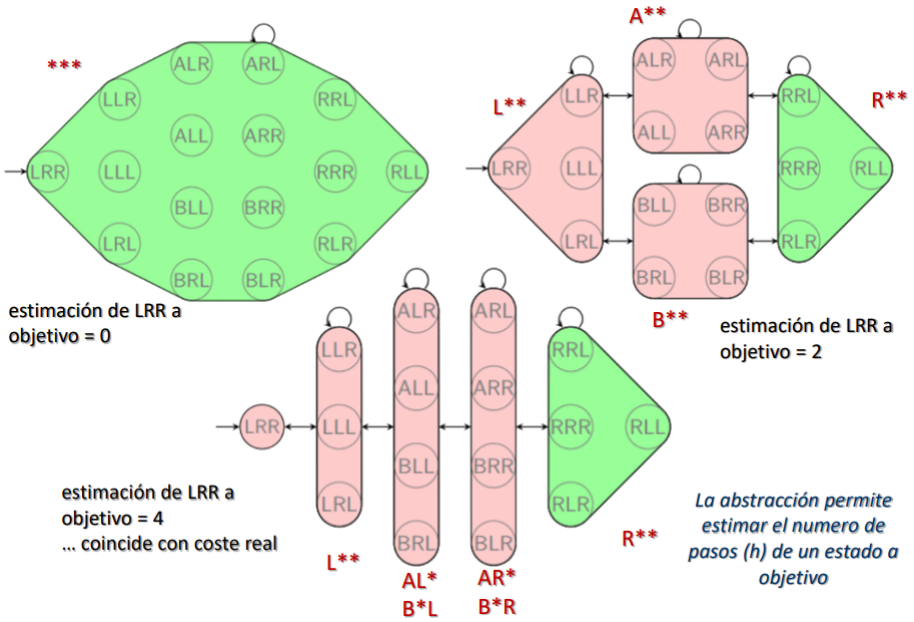

Figura 2.5: Ejemplo de abstracciones en un grafo de transición.
Imagen recuperada de: material docente y transparencias de poliformat de la asignatura Técnicas, Entornos y Aplicaciones de Inteligencia Artificial

2.2.1.3. Heurística por *landmarks*

La heurística de *landmarks* surgió inicialmente como una técnica para descomponer problemas y establecer secuencias lógicas en los planificadores. Se introdujo la noción de órdenes razonables, que establece que un par de objetivos A y B pueden ser ordenados de tal manera que B se logre antes que A si no es posible alcanzar un estado en el que ambos, A y B, sean verdaderos a partir de un estado en el que solo A es verdadero, sin tener que eliminar temporalmente A. En situaciones como esta, es lógico y eficiente lograr B antes que A para evitar esfuerzos innecesarios.

Un elemento clave en esta heurística es el concepto de *landmark* o hito, que debe ser verdadero en algún momento en cualquier camino de la solución para la tarea de planificación dada [13]. En el contexto de movilidad urbana, se puede aplicar esta heurística considerando un vehículo que inicia su trayecto desde un punto determinado. Durante su recorrido, el vehículo sigue una secuencia constante: siempre gira a la derecha en una calle predeterminada y luego se encuentra con una variedad de caminos que conducen a su lugar de trabajo. Para este ejemplo, los *landmarks* podrían ser la partida desde el punto inicial, la secuencia de giros a la derecha en la calle predeterminada y finalmente, la llegada al lugar de trabajo.

Visualmente, esto se puede representar de manera gráfica, utilizando <A>, <B>y <C>como *landmarks*, y los diferentes giros y elecciones que no se consideran *landmarks* como números como se muestra en la Figura 2.6. Esta representación gráfica permite comprender cómo los *landmarks* se convierten en puntos cruciales que deben ser alcanzados en cualquier plan válido para la movilidad del vehículo.

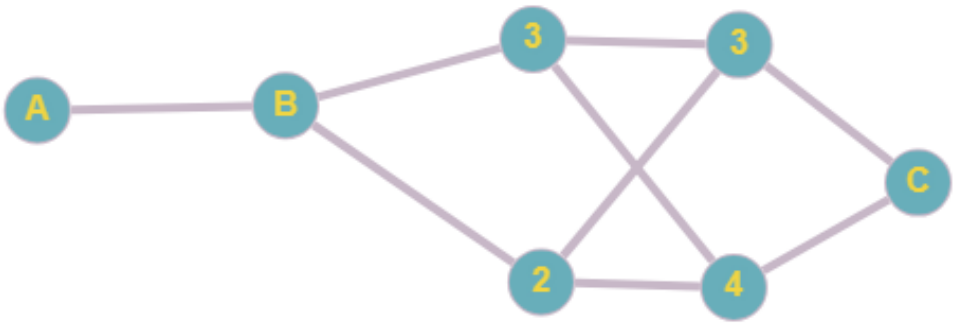

Figura 2.6: Ejemplo de landmarks.

2.2.2. PDDL

El lenguaje de definición de dominio de planificación (PDDL, por sus siglas en inglés) es un intento de estandarizar los lenguajes de planificación de inteligencia artificial (IA). Fue desarrollado por primera vez por Drew McDermott y

sus compañeros en 1998 (inspirados en STRIPS [1] y ADL [2], entre otros), principalmente para hacer posible la Competencia Internacional de Planificación (IPC) de 1998/2000, y luego evolucionó con cada competencia. La estandarización proporcionada por PDDL tiene el beneficio de hacer que la investigación sea más reutilizable y fácilmente comparable, aunque a costa de cierta potencia expresiva en comparación con los sistemas específicos de dominio.[6]

2.2.2.1. Modelización de un problema de planificación en PDDL

La modelización de un problema de planificación en PDDL implica describir de manera formal y estructurada los elementos clave de un escenario específico. Esto incluye definir el estado inicial del sistema, las posibles acciones que pueden ocurrir y las metas que se deben lograr. PDDL proporciona un estándar para expresar estas descripciones, lo que facilita la aplicación de técnicas de planificación automatizada. Un problema en PDDL se caracteriza por tener dos tipos de archivos:

- **Dominio PDDL**: Un archivo de dominio en PDDL define los aspectos 'universales' de un problema. Básicamente, estos son los aspectos que no cambian independientemente de la situación específica que estemos tratando de resolver. En PDDL, esto consiste principalmente en los tipos de objetos, predicados y acciones que pueden existir dentro del modelo[7]. En la Figura 2.7 podemos observar un dominio de ejemplo, en este caso modelando un problema de carga y descarga de paquetes con camiones, con diferentes acciones para mover, descargar y cargar el camión. En este ejemplo, los aspectos universales son:
  - **Tipos de objetos:** utilizamos los tipos para restringir qué objetos pueden formar parte de los parámetros de una acción. En el ejemplo de la Figura 2.7, contamos con dos tipos de objetos denominados *'location'* y *'locatable'*. Si nos fijamos nos daremos cuenta de que el tipo de objeto *'locatable'* es a su vez el tipo de objeto de *'truck'* y *'obj'*, haciendo esto estamos definiendo subtipos de objetos, diciendo que estos deben ser localizables y poder restringir aun más el tipo de objetos en los parámetros de las acciones.
  - **Predicados:** dan información proposicional sobre el problema, pueden ser verdaderos o falsos en cualquier momento de un plan y, cuando no se declaran, se asume que son falsos. Para el ejemplo de la Figura 2.7, el dominio cuenta con tres predicados, uno de ellos para expresar donde se encuentra un objeto de tipo *'locatable'*, otro para saber si un objeto de tipo *'obj'* se encuentra dentro de un camión y por último un predicado llamado *'link'* para determinar si dos localizaciones están unidas.
  - **Acciones:** definen una transformación en el estado del mundo. Esta transformación suele ser una acción que podría llevarse a cabo en la ejecución del plan. En el ejemplo de la Figura 2.7 tenemos tres acciones una para cargar el camión, otra para descargar el camión y una última

[1] https://es.wikipedia.org/wiki/STRIPS
[2] https://en.wikipedia.org/wiki/Action_description_language

para mover el camión entre localizaciones. Las acciones a su vez se pueden dividir en diferentes secciones, estas secciones son:

1. **Parámetros:** se definen los objetos que la acción va a usar, con un nombre para los diferentes objetos que usaremos. En la Figura 2.7 la acción *'LOAD-TRUCK'*, tiene como parámetros un objeto denominado *'?obj'*, otro objeto denominado *'?truck'* y finalmente un objeto denominado *'?loc'* que se usarán posteriormente en la sección de precondiciones y en la sección de efectos.
2. **Precondiciones:** son las condiciones que se deben cumplir para que la acción se pueda llevar a cabo. Las precondiciones para la acción *'LOAD-TRUCK'* de la Figura 2.7 serían *'(at ?truck ?loc)* y *'(at ?obj ?loc)'*. haciendo referencia a que el objeto *'?truck'* debe encontrarse en la misma localización *'?loc'* que el objeto *'?obj'* para así poder cargarlo.
3. **Efectos:** cuando una acción se lleva a cabo, los efectos determinan qué predicados se añaden al nuevo estado y cuáles se eliminan. Continuando con la misma acción de *'LOAD-TRUCK'*, sus efectos son, *'(not (at ?obj ?loc))'* y *'(in ?obj ?truck)'*, añadiendo al nuevo estado el predicado *(in ?obj ?truck)* que se traduce en que el objeto *?obj* se encuentra dentro del camión y eliminando el efecto *'(not (at ?obj ?loc))'*, que se traduce como que el objeto *?obj* ya no se encuentra en la localización *'?loc'*

```
(define (domain driverlog)
(:requirements :typing :numeric-fluents)
Show hierarchy
(:types
  location locatable - object
  truck obj - locatable
)

(:predicates
  (at ?obj - locatable ?loc - location)  ; `?obj` is at location `?loc`
  (in ?obj1 - obj ?t - truck) 1👁 1☑ 1☒ ; package `?obj` is in a truck `?t`
  (link ?x ?y - location) 1👁 ; there is a _road_ between `?x` and `?y`
)

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

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

(:action DRIVE-TRUCK
  :parameters (?truck - truck ?loc_from - location ?loc_to - location)
  :precondition (and (at ?truck ?loc_from) (link ?loc_from ?loc_to))
  :effect (and (not (at ?truck ?loc_from)) (at ?truck ?loc_to))
)

)
```

Figura 2.7: Ejemplo de un dominio sobre carga y descarga de paquetes con camiones

- **Problema PDDL** : El problema en PDDL conforma la otra mitad de un problema de planificación. En el dominio expresamos los aspectos globales y 'mundanos' de un problema, como qué acciones podemos realizar y qué tipos de objetos existen en el mundo en el que estamos planeando.

  El problema luego solidifica esta expresión al definir exactamente qué objetos existen y qué es cierto acerca de ellos, y finalmente cuál es el objetivo final, en qué estado queremos que esté el mundo una vez que se haya completado el plan.[7] En la Figura 2.8 podemos observar el problema asociado al dominio de la Figura 2.7. Podemos dividir el problema en tres partes principales:

  1. **Objetos:** esta sección nos permite definir los objetos que existirán en nuestro problema. Cada objeto debe tener un nombre único y estar asociado a un tipo de objeto especificado en el dominio. En la Figura 2.8 se definen cuatro tipos de localizaciones nombrados como *'lot0'*, *'lot1'*, *'bldg0'* y *'bldg1'*, un tipo de camión con nombre *'truck1'* y un tipo de objeto llamado *'package1'* para diferenciarlos.
  2. **Estado inicial:** en él se definen todos los predicados que consideramos verdaderos al inicio de nuestro problema. Para el ejemplo de la Figura 2.8 se determinan la localización del camión y el paquete, así como que localizaciones estan conectadas entre sí.
  3. **Objetivos:** por último tenemos los objetivos, que son todos los predicados que se deben cumplir para que un plan se considere una solución. La Figura 2.8 tiene como objetivos *'(at package1 bldg1)'* y *'(at truck1 lot0)'*, que se traduce en que el paquete debe estar en la localización *'bldg1'* y el camión debe estar en la localización *'lot0'*.

```
(define (problem driverlog-problem)
(:domain driverlog)
Show hierarchy
(:objects
    lot0 lot1 bldg0 bldg1 - location
    truck1 - truck
    package1 - obj
)
View
(:init
    (at truck1 lot1)

    (at package1 bldg0)

    (link lot0 lot1)
    (link lot1 lot0)

    (link lot1 bldg0)
    (link bldg0 lot1)

    (link bldg0 bldg1)
    (link bldg1 bldg0)

    (link bldg1 lot0)
    (link lot0 bldg1)
)

(:goal (and
    (at package1 bldg1)
    (at truck1 lot0)
))
)
```

Figura 2.8: Ejemplo de un problema sobre carga y descarga de paquetes con camiones

2.2.3. Planificador

Una vez definidos el dominio y el problema es hora de obtener un plan, y para esto es necesario un planificador. La función principal de un planificador es recibir una descripción formal del dominio del problema y una instancia específica del problema, y luego generar un plan que, cuando se ejecuta en el entorno del problema, lleve desde un estado inicial a un estado objetivo deseado. Existen una gran variedad de planificadores que están disponibles públicamente, en su mayoría para ser ejecutados con el sistema operativo Linux. Algunos ejemplos de estos planificadores son:

- **LPG:** es un planificador que se basa en métodos de búsqueda local y en el uso de grafos de planificación. Su esquema de búsqueda está inspirado en Walksat [3], un método eficiente para resolver problemas SAT[4], y se centra en el espacio de búsqueda conformado por 'gráficos de acciones'. Estos gráficos son subconjuntos particulares del grafo de planificación que representan planes parciales.

  Los pasos de búsqueda en LPG implican modificaciones específicas en el grafo, transformando un gráfico de acciones en otro. Este enfoque aprovecha una representación compacta del grafo de planificación para definir el vecindario de búsqueda. La evaluación de los elementos del vecindario se realiza mediante una función parametrizada, donde los parámetros ponderan diferentes tipos de inconsistencias presentes en el plan parcial actual. Durante la búsqueda, estos parámetros se evalúan dinámicamente mediante multiplicadores de Lagrange discretos.

  La función de evaluación emplea heurísticas como el 'costo heurístico de búsqueda' y el 'costo de ejecución heurística' para satisfacer una precondición, posiblemente numérica. Las duraciones de las acciones y las cantidades numéricas, como el consumo de combustible, se representan en los gráficos de acciones y se consideran en la función de evaluación.

  En dominios temporales, las acciones se ordenan utilizando un 'grafo de precedencia', el cual se actualiza durante la búsqueda y tiene en cuenta las relaciones mutex del grafo de planificación. Estas relaciones reflejan la exclusividad entre dos acciones, indicando que no pueden ocurrir simultáneamente.[8]

- **Mips-xxl**: es un software de planificación desarrollado por Stefan Edelkamp, Shahid Jabbar y Mohammed Nazih. Este programa es compatible con PDDL. El término 'XXL' en su nombre se debe a su capacidad para explorar espacios de estado mucho más grandes que la memoria RAM disponible. Implementa Algoritmos de Planificación de Memoria Externa que utilizan el disco duro de manera eficiente.[10]

- **OPTIC:** es un planificador temporal diseñado para problemas en los que el coste del plan está determinado por preferencias o costes de colección de objetivos dependientes del tiempo. Estos problemas surgen en diversas

[3] https://en.wikipedia.org/wiki/WalkSAT
[4] https://es.wikipedia.org/wiki/Problema_de_satisfacibilidad_booleana

situaciones, desde la programación de la entrega de productos perecederos hasta la coordinación de actividades de cumplimiento de pedidos en almacenes.[11]

- **SGPlan:** divide un problema extenso de planificación en subproblemas, cada uno con su propio subobjetivo, y resuelve esas soluciones inconsistentes utilizando su condición extendida de punto de silla. SGPlan generaliza la partición de subobjetivos para aprovechar la localidad de restricciones debido a las nuevas características en PDDL3. Desarrolla técnicas para resolver restricciones violadas, optimizar preferencias de objetivos y lograr subobjetivos en una representación multinivel. La implementación actual de SGPlan utiliza un planificador Metric-FF[5] modificado para la planificación básica.[12]

En la Figura 2.9 podemos observar un ejemplo de plan obtenido por el planificador LPG usando el dominio de la Figura 2.7 y el problema de la Figura 2.8, al ser un dominio y problema muy simple el planificador LPG solo encuentra una solución óptima, que además cuenta con únicamente cinco acciones.

```
0:   (DRIVE-TRUCK TRUCK1 LOT1 BLDG0) [1]
1:   (LOAD-TRUCK PACKAGE1 TRUCK1 BLDG0) [1]
2:   (DRIVE-TRUCK TRUCK1 BLDG0 BLDG1) [1]
3:   (UNLOAD-TRUCK PACKAGE1 TRUCK1 BLDG1) [1]
4:   (DRIVE-TRUCK TRUCK1 BLDG1 LOT0) [1]
```

Figura 2.9: Ejemplo de un plan devuelto por el planificador LPG

2.2.4. Validación/Val

La validación de planes es un componente fundamental en la planificación, asegurando que los planes generados sean coherentes y factibles. El Plan Validation Tool (VAL)[6] ha surgido como una herramienta crucial en este ámbito, inicialmente desarrollada para respaldar la Competencia Internacional de Planificación y posteriormente ampliada para abordar desafíos específicos en la planificación de operaciones espaciales. Este software ha demostrado ser esencial al garantizar la coherencia y viabilidad de los planes generados.

El VAL desempeña un papel crucial en situaciones donde la complejidad de las funciones no permite que los sistemas automáticos realicen la planificación inicial o cuando las metas son demasiado complejas para expresarse fácilmente en un lenguaje de planificación. Facilita la interacción entre humanos y máquinas en la construcción de planes, formando parte de lo que se conoce como planificación mixta. Además de validar planes, el VAL también ofrece orientación para corregir planes defectuosos, fomentando así un ciclo iterativo de mejora en la planificación.[2]

[5] https://fai.cs.uni-saarland.de/hoffmann/ff.html
[6] https://nms.kcl.ac.uk/planning/software/val.html

El VAL se ha convertido en un pilar en el campo de la planificación al permitir una colaboración efectiva entre humanos y sistemas automáticos, asegurando que los planes desarrollados sean robustos, alcanzables y adaptativos a medida que evolucionan las condiciones y requisitos.

2.2.5. Adaptación y Reparación

La discusión entre adaptar planes existentes y generar nuevos planes es un tema de constante reflexión en la literatura dedicada a la planificación y ejecución de acciones. Autores prominentes como Nebel y Koehler (1995) junto con Gerevini y Serina (2000) han profundizado en las ventajas y desventajas tanto teóricas como empíricas asociadas con estas dos aproximaciones. La reparación implica ajustar un plan existente a un nuevo contexto con el objetivo de minimizar las perturbaciones, mientras que la replanificación involucra la creación de un nuevo plan desde cero, sin tomar en cuenta la estabilidad del plan anterior [16].

La reparación se destaca como un método popular que permite a un agente que ha enfrentado un fallo generar rápidamente un nuevo plan para alcanzar sus objetivos. Sin embargo, este enfoque genera una nueva solución sin considerar completamente el plan existente. Por otro lado, la reparación de planes busca corregir el plan existente, intentando preservar sus elementos y estructura, adaptándolo al nuevo contexto y causando perturbaciones mínimas.

En teoría, se argumenta que, en el peor de los casos, modificar un plan existente no es más eficiente que realizar una replanificación completa desde el estado actual. No obstante, en la práctica, numerosas investigaciones previas (Bechon et al., 2020; Chen et al., 2020; van der Krogt y de Weerdt, 2005; Gerevini y Serina, 2000) sugieren que la reparación de planes puede ser más efectiva que la replanificación, especialmente cuando la duración del plan (es decir, la duración total del plan) no es el principal criterio a optimizar [4].

CAPÍTULO 3
Diseño de la solución

En este capítulo, abordaremos la definición del dominio y el problema en PDDL, acompañados de un ejemplo con el plan generado mediante el planificador LPG. A continuación, exploraremos el enfoque para validar un plan, detallando los pasos para corregirlo en caso de invalidez o, en su defecto, replanificarlo por completo, cabe recalcar que el enfoque de validación y su posible corrección o replanificación esta diseñado para ser genérico y no exclusivo del dominio, problema y plan que se han desarrollado para el proyecto y que se van a mostrar en este capitulo. Esta sección servirá como punto de partida para comprender el proceso completo, brindando una visión detallada de cómo abordaremos la validación, reparación o replanificación de los planes generados. Concluiremos esta sección presentando un ejemplo reducido, lo que facilitará una explicación más clara y comprensible del proceso en acción.

3.1 Modelización del dominio y el problema

Para abordar la solución propuesta, es necesario el desarrollo de un dominio y problema, aplicando los conceptos delineados en el capítulo 2 sobre la modelización en PDDL. En este contexto, dirigiremos nuestra atención hacia un escenario específico: el de los taxis y autobuses autónomos.

Este proceso de modelización implica la definición precisa de acciones, predicados y reglas que gobiernan el comportamiento y las interacciones de estos vehículos autónomos en un entorno determinado. A través de la representación formal en PDDL, buscamos capturar las dinámicas generales que caracterizan la movilidad autónoma.

El dominio diseñado incorporará acciones clave, como 'moverse a un destino', 'recoger pasajero' y 'dejar pasajero', mientras que los predicados reflejarán el estado dinámico del entorno. En paralelo, el problema específico que plantearemos dentro de este dominio contendrá instancias concretas de vehículos, destinos y pasajeros.

3.2 Primer dominio propuesto

En el contexto inicial del proyecto, se consideró un primer dominio para la implementación. Sin embargo, durante la planificación, surgieron limitaciones que nos llevaron a descartar esta primera propuesta. Fue necesario modificar este enfoque y optar por otro dominio, el cual será explicado en detalle en la siguiente sección.

En este primer dominio, se persiguió una representación del mundo real. Los taxis y autobuses estaban sujetos a restricciones de carga que influían en sus recorridos. Se instalaron estaciones de carga para recargar los vehículos, y las distancias entre ubicaciones tenían un impacto directo en la capacidad de carga de los taxis y autobuses. Además, se establecieron límites de pasajeros para estos vehículos. Se utilizó una métrica específica para minimizar la cantidad de recargas que debían hacer los taxis y autobuses. Las acciones se caracterizaban como durativas, lo que implicaba que requerían un tiempo definido para completarse. En conjunto, se logró obtener una representación más realista del entorno.

Para empezar a definir este dominio en PDDL lo primero es definir todos los objetos que utilizaremos en las funciones y acciones del dominio, en nuestro caso hemos definido un total de cuatro objetos distintos: *taxi*, *bus*, *location* y *passenger*, los podemos observar en la Figura 3.1.

```
(:types
  taxi - object
  bus - object
  location - object
  passenger - object
)
```

Figura 3.1: Tipos de objeto del dominio

Los predicados, como se han detallado en el capítulo 2, proveen información proposicional crucial para nuestro problema. En este nuevo dominio, se han definido un total de seis predicados, detallados en la Figura 3.2. Aunque no profundizaremos en cada uno de ellos en esta instancia, ya que en la versión final del dominio se han agregado algunos predicados nuevos y se han eliminado otros.

En términos generales, estos predicados nos ofrecen información sobre la ubicación de taxis y autobuses, la localización de pasajeros en taxis, autobuses o en una localización, la ocupación de taxis o áreas específicas, la existencia de rutas entre distintas localizaciones, la conectividad entre dos localizaciones distintas, así como la ubicación de puntos de carga destinados a taxis y autobuses.

```
(:predicates (at ?x - (either taxi bus) ?l - location)
             (in ?p - passenger ?x - (either location taxi bus))
             (occupied ?x - (either taxi location))
             (route ?l1 - location ?l2 - location)
             (connected ?l1 - location ?l2 - location)
             (charging-point ?l - location)    )
```

Figura 3.2: Predicados del primer dominio propuesto

Aunque en el capítulo 2 no se abordaron las funciones en el contexto de PDDL, en la primera versión del dominio estas jugaron un papel importante. Las funciones representan información numérica dentro del problema, como la cantidad de batería de un taxi o la cantidad de pasajeros en un autobús. Se definieron un total de diez funciones, detalladas en la Figura 3.3. Al igual que con los predicados, no profundizaremos en cada una de ellas en esta instancia, aunque se explicarán de manera general.

Estas funciones contienen datos numéricos cruciales sobre la carga de batería de taxis y autobuses, ya que en el mundo real, la capacidad de carga de un vehículo es fundamental para determinar sus posibles trayectos. La distancia entre dos puntos geográficos también es vital, ya que junto con la carga del vehículo, determina su capacidad para alcanzar esas ubicaciones.

Asimismo, se considera la cantidad actual de pasajeros en un autobús en comparación con su capacidad máxima. En la realidad, un autobús tiene límites claros en cuanto al número de pasajeros que puede transportar.

Otra función fundamental es la duración temporal de las acciones, ya que en el mundo real, las actividades no ocurren de manera instantánea. Cada acción conlleva un tiempo específico para su ejecución, lo que influye en la eficiencia y planificación de rutas en el contexto del problema.

Por último, se incorpora una función adicional que servirá como métrica para minimizar el costo de las recargas en taxis y autobuses. El objetivo es optimizar la eficiencia de recarga de estos vehículos, buscando reducir al máximo los gastos asociados a estas operaciones. Esto contribuirá a diseñar estrategias más eficientes y económicas para mantener los vehículos en funcionamiento sin incurrir en costos excesivos de recarga.

El siguiente paso en el modelo del dominio son las acciones, para este dominio se van a mostrar dos acciones de ejemplo para mostrar la complejidad de las mismas y como se relacionan con el mundo real y los predicados y funciones mostradas anteriormente. Las acciones se muestran en la Figura 3.4, podemos observar las acciones *Drive_taxi* y *Charge_taxi*, que son las que más cambian con respecto al dominio que se usa finalmente.

En la acción *Drive_taxi*, la duración se establece considerando la distancia entre las dos ubicaciones y dividiéndola entre dos. Esta aproximación no tiene en cuenta la velocidad del taxi, por lo que se divide la distancia para evitar que la acción sea demasiado extensa. Además, como precondición se tiene en cuenta la distancia entre las localizaciones y la carga del vehículo para evaluar la viabilidad

```
(:functions (charge ?x - (either taxi bus))
            (distance ?l1 - location ?l2 - location)
            (passengers-bus ?b - bus)
            (all-recharge-cost)
            (pickup-time)
            (drop-off-time)
            (charge-cost)
            (amount-of-charge)
            (recharge-time)
            (max-passengers-bus)
            )
```

Figura 3.3: Funciones del primer dominio propuesto

del trayecto, todo ello para intentar que nuestras acciones sean una representación lo más fiel posible al mundo real.

Al completarse la acción, la carga del taxi disminuye en la cantidad equivalente a la distancia recorrida entre los dos puntos. Esta reducción representa el efecto de la acción en la carga del vehículo.

La acción *Charge_taxi* fue eliminada en la versión final del dominio. Esta acción involucraba recargar la batería del taxi en una localización equipada con un punto de carga. Se consideraba que el taxi debía estar vacío, reflejando la práctica común en el mundo real, donde los taxis se recargan típicamente cuando no transportan pasajeros y la localización está desocupada.

Una localización con un punto de carga presentaba una única estación de carga y solo un taxi podía recargarse a la vez. Al completarse la acción, la batería del taxi se recargaba a una cantidad específica definida en el problema. Además, el costo asociado con la carga se agregaba a nuestra métrica a minimizar, representando así el impacto económico de estas recargas en la eficiencia del sistema.

Otra acción clave es *Pickup_bus*, que se describe en la Figura 3.5. A diferencia de la acción final utilizada, esta acción incluye una precondición que verifica si la cantidad actual de pasajeros en el autobús es menor que su capacidad máxima. Al completarse esta acción, el recuento de pasajeros en el autobús aumenta en uno, reflejando el número total de pasajeros a bordo. Esta implementación se enfoca en replicar fielmente la realidad, donde los autobuses tienen una capacidad limitada para pasajeros.

El principal obstáculo que llevó a descartar este dominio fue la falta de *parsers* disponibles actualmente para convertir nuestro dominio a Python. Ninguno de los *parsers* identificados hasta ahora cumplía con todos los requisitos necesarios, ya que se centraban en la planificación clásica y no admitían funciones ni acciones durativas.

Ante esta limitación, la decisión fue adaptar el dominio existente para simplificarlo y hacerlo compatible con los *parsers* actuales. Esta adaptación fue necesaria para trabajar con las herramientas disponibles, aunque implicó renunciar a ciertas funcionalidades que mejoraban la fidelidad del dominio con respecto al mundo real.

```
(:durative-action drive_taxi
 :parameters (?t - taxi ?l1 ?l2 - location)
 :duration (= ?duration (/ (distance ?l1 ?l2) 2))
 :condition (and (at start (at ?t ?l1))
                 (at start (connected ?l1 ?l2))
                 (at start (>= (charge ?t)
                               (distance ?l1 ?l2))))

 :effect (and (at start (not (at ?t ?l1)))
              (at end (at ?t ?l2))
              (at end (decrease (charge ?t)
                                (distance ?l1 ?l2 )))))

(:durative-action charge_taxi
 :parameters (?t - taxi ?l - location)
 :duration (= ?duration (recharge-time))
 :condition (and (over all (at ?t ?l))
            (at start(charging-point ?l))
            (at start (not (occupied ?t)))
            (at start (not (occupied ?l))))
 :effect (and (at end (increase (charge ?t) (amount-of-charge)))
              (at end (increase (all-recharge-cost) (charge-cost)))
              (at start(occupied ?t))
              (at start (occupied ?l))
              (at end (not(occupied ?t)))
              (at end (not(occupied ?l)))))
```

Figura 3.4: Ejemplo de acciones del primer dominio propuesto

```
(:durative-action pickup_bus
 :parameters (?p - passenger ?b - bus ?l - location)
 :duration (= ?duration (pickup-time))
 :condition (and (at start (in ?p ?l))
                 (at start (< (passengers-bus ?b) (max-passengers-bus)))
                 (over all (at ?b ?l)))
 :effect (and (at start (not (in ?p ?l)))
              (at end (increase (passengers-bus ?b) 1))
              (at end (in ?p ?b))))
```

Figura 3.5: Acción *Pickup_bus* del primer dominio propuesto

3.3 Dominio empleado

Este nuevo dominio es una variante modificada del presentado anteriormente. A pesar de los cambios, uno de los pocos elementos que se ha mantenido inalterado son los objetos definidos, ya que son igualmente aplicables a ambos dominios. Estos objetos son consistentes y se mantienen idénticos en ambas versiones, como se puede apreciar en la Figura 3.1

Para continuar con la definición del dominio, se han establecido un total de ocho predicados, en comparación con los seis definidos en el primer dominio. Algunos han sido eliminados de los definidos previamente, otros se han mantenido, mientras que se han añadido nuevos, los cuales se detallan en la Figura 3.6. A continuación, procederemos a explicar cada uno de ellos:

- ***At_taxi ?t - taxi ?l - location***, este predicado hace referencia a los objetos *taxi* y *location*, cuando un taxi este en una localización este predicado estará dentro de un estado del problema. Un ejemplo seria *at_taxi taxi1 loc1*, lo que quiere decir que el taxi 1 se encuentra en la localización 1 en el estado actual del problema.

- ***At_bus ?b - bus ?l - location***, similar al predicado anterior, con la única diferencia de que esta hace referencia a un objeto *bus* en vez de a un objeto *taxi*.

- ***Passenger_at ?p - passenger ?l - location***, similar a los anteriores predicados, aunque esta vez haciendo referencia a un objeto *passenger*, necesario para cuando queremos subir o bajar a un pasajero de un taxi o bus.

- ***Connected ?l1 - location ?l2 - location***, este predicado utiliza dos objetos *location* para determinar si dos localizaciones estan conectadas y por tanto un taxi puede moverse entre ellas. Un ejemplo sería *connected loc3 loc7*, o lo que es lo mismo, la localización 7 esta conectada con la localización 8.

- ***On_board_taxi ?p - passenger ?t - taxi***, para saber cuando un pasajero sube a un taxi necesitamos un predicado, que es justamente la información que nos proporciona este predicado, teniendo como ejemplo *on_board_taxi passenger1 taxi4* quiere decir que el pasajero 1 se encuentra subido al taxi 4.

- ***On_board_bus ?p - passenger ?b - bus***, es un predicado igual al anterior con la única diferencia de que éste hace referencia al objeto *bus*.

- ***Route ?l1 - location ?l2 - location***, para diferenciar los buses de los taxis, los buses no necesitan que dos localizaciones estén conectadas con el predicado *connected*, en su caso los autobuses tienen un predicado propio de ruta que les permite ir a cualquier destino.

- ***Empty_taxi ?t - taxi***, este predicado es necesario para que nos cercioremos de que solo un pasajero puede subir al taxi.

En este dominio centrado en la planificación clásica, no se emplean funciones, ya que este enfoque prescinde de considerar funciones numéricas. Esta limitación

```
(:predicates
  (at_taxi ?t - taxi ?l - location)
  (at_bus ?b - bus ?l - location)
  (passenger_at ?p - passenger ?l - location)
  (connected ?l1 - location ?l2 - location)
  (on_board_taxi ?p - passenger ?t - taxi)
  (on_board_bus ?p - passenger ?b - bus)
  (route ?l1 - location ?l2 - location)
  (empty_taxi ?t - taxi)
)
```

Figura 3.6: Predicados del dominio

nos restringe considerablemente en la capacidad de lograr una representación más fiel del mundo real. Lamentablemente, como se ha explicado previamente, esta limitación fue un sacrificio necesario para adaptarnos a las herramientas actualmente disponibles en línea.

Concluida la definición de los predicados, pasamos a las acciones. En este dominio, contamos con un total de seis acciones: tres asociadas a taxis y otras tres relacionadas con autobuses. En nuestro primer dominio, teníamos un total de ocho acciones, cuatro para cada tipo de vehículo. En este caso, se eliminaron las acciones *Charge_taxi* y *Charge_bus* debido a su dependencia con las funciones, lo que las hace incompatibles con la planificación clásica.

Las acciones empleadas en este dominio se muestran en la Figura 3.7. A continuación, explicaremos detalladamente cada acción, sus parámetros, precondiciones y efectos:

- ***Drive_taxi***: Esta acción gestiona los movimientos entre distintas localizaciones de un taxi, verificando que el taxi se encuentre en la posición de origen y que la localización de destino esté conectada a la de origen. Los parámetros de esta acción son *?t*, *?src* y *?to*, de tipo *taxi*, *location* y *location* respectivamente. Tiene las siguientes precondiciones:
  - *(at_taxi ?t ?src)*: El taxi debe estar en la localización de origen para moverse.
  - *(connected ?src ?to)*: Asegura que las dos localizaciones estén conectadas, permitiendo al taxi llegar a la segunda localización.

  Y los siguientes efectos:
  - *(not(at_taxi ?t ?src))*: Elimina el predicado para indicar que el taxi ya no se encuentra en la localización original.
  - *(at_taxi ?t ?to)*: Indica que el taxi se encuentra ahora en la localización destino.
- ***Pickup_taxi***: Esta acción define el proceso de subir un pasajero a un taxi, verificando si el pasajero y el taxi se encuentran en la misma ubicación y si

el taxi está vacío. Posteriormente, actualiza la ubicación del pasajero al taxi y marca el taxi como ocupado. Los parámetros de esta acción son *?t*, *?p*, *?l* de tipo *taxi*, *passenger*, *location*, respectivamente. Cuenta con las siguientes precondiciones:

- *(at_taxi ?t ?l)*: Determina la ubicación actual del taxi.
- *(passenger_at ?p ?l)*: Verifica si las ubicaciones del taxi y el pasajero coinciden, utilizando el objeto *?l* de tipo *location*.
- *(empty_taxi ?t)*: Diferencia un taxi con pasajero de uno vacío, permitiendo que el taxi suba a un pasajero.

Y los siguientes efectos:

- *(not(passenger_at ?p ?l))*: Indica que el pasajero ya no se encuentra en la ubicación original.
- *(on_board_taxi ?p ?t)*: Establece que el pasajero está ahora en el taxi.
- *(not(empty_taxi ?t))*: Marca el taxi como ocupado, impidiendo que suba más pasajeros hasta que esté vacío nuevamente.

- ***Dropoff_taxi***: Esta acción define el proceso de bajada de un pasajero de un taxi. Primero verifica en qué localización se encuentra el taxi y si tiene a un pasajero subido. Si el pasajero está en el taxi, lo retira de este y lo coloca en la localización del taxi; luego, marca el taxi como vacío para que pueda recibir un nuevo pasajero. Los parámetros son los mismos que los de la acción anterior. Esta acción contiene las siguientes precondiciones:

  - *(at_taxi ?t ?l)*: Verifica la localización actual del taxi.
  - *(on_board_taxi ?p ?t)*: Comprueba si el pasajero está en el taxi.

  Y contiene los siguientes efectos:

  - *(not(on_board_taxi ?p ?t))*: Indica que el pasajero ya no está en el taxi.
  - *(passenger_at ?p ?l)*: Coloca al pasajero en la misma localización que el taxi.
  - *(empty_taxi ?t)*: Marca el taxi como vacío, listo para recibir un nuevo pasajero.

Las acciones del autobús son muy similares a las del taxi, con pequeñas modificaciones que son las que explicaremos a continuación:

- ***Drive_bus***: Esta primera acción, al igual que la acción inicial del taxi, sirve para mostrar los movimientos del autobús por las diferentes localizaciones, con la diferencia de que para esta usamos un objeto *bus* en vez de un objeto *taxi* y tenemos una precondición distinta *(route ?src ?to)* en vez de *(connected ?src ?to)*. Se hace esta diferencia para que los autobuses puedan seguir diferentes trayectos en comparación con los taxis y tengan más posibilidades, quedando los efectos de igual manera que su acción igualitaria pero utilizando el objeto *bus*.

- ***Pickup_bus***: Esta acción mantiene similitudes con su contraparte relacionada al objeto 'taxi'. No obstante, se omite una de las precondiciones presentes en *pickup_taxi*: *(empty_taxi ?t)*. Como hemos discutido previamente, no disponemos de un predicado exclusivo para los autobuses. Esta omisión se realiza para permitir que un autobús no tenga restricciones en cuanto a los pasajeros a bordo. Dado que estamos empleando un modelo de planificación clásica, no podemos limitar el número de pasajeros mediante una función, como se hizo en el primer dominio propuesto.

- ***Dropoff_bus***: Para finalizar con las acciones, encontramos la acción *dropoff_bus*, que difiere de la acción *dropoff_taxi* únicamente en la eliminación de uno de sus efectos. Similar a la acción anterior, donde eliminamos una precondición para no limitar la capacidad del autobús, aquí prescindimos de un efecto que indicaba si el autobús estaba completo o no. En otras palabras, el efecto que estamos eliminando sería el equivalente a *(empty_taxi ?t)*.

```
(:action drive_taxi
  :parameters (?t - taxi ?src - location ?to - location)
  :precondition (and (at_taxi ?t ?src) (connected ?src ?to))
  :effect (and (not (at_taxi ?t ?src)) (at_taxi ?t ?to))
)
(:action pickup_taxi
  :parameters (?t - taxi ?p - passenger ?l - location)
  :precondition (and (at_taxi ?t ?l) (passenger_at ?p ?l) (empty_taxi ?t))
  :effect (and (not (passenger_at ?p ?l)) (on_board_taxi ?p ?t) (not(empty_taxi ?t)))
)
(:action dropoff_taxi
  :parameters (?t - taxi ?p - passenger ?l - location)
  :precondition (and (at_taxi ?t ?l) (on_board_taxi ?p ?t))
  :effect (and (not (on_board_taxi ?p ?t)) (passenger_at ?p ?l) (empty_taxi ?t))
)

(:action drive_bus
  :parameters (?b - bus ?src - location ?to - location)
  :precondition (and (at_bus ?b ?src) (route ?src ?to))
  :effect (and (not (at_bus ?b ?src)) (at_bus ?b ?to))
)
(:action pickup_bus
  :parameters (?b - bus ?p - passenger ?l - location)
  :precondition (and (at_bus ?b ?l) (passenger_at ?p ?l) )
  :effect (and (not (passenger_at ?p ?l)) (on_board_bus ?p ?b))
)
(:action dropoff_bus
  :parameters (?b - bus ?p - passenger ?l - location)
  :precondition (and (at_bus ?b ?l) (on_board_bus ?p ?b))
  :effect (and (not (on_board_bus ?p ?b)) (passenger_at ?p ?l))
)
```

Figura 3.7: Acciones del dominio

3.4 Primer problema propuesto

Ahora nos adentramos en la definición del problema específico, que inicialmente se planteó de manera similar al dominio. Sin embargo, debido a las mismas limitaciones y razones mencionadas anteriormente, se requirió modificarlo, como se detallará en la siguiente sección. En esta etapa, se destacarán las características principales que lo distinguían del problema utilizado finalmente. Afortunadamente, a diferencia del dominio, gran parte del plan pudo conservarse, con excepción de los elementos que mencionaremos a continuación.

Para definir el problema, fue necesario inicializar todas las funciones establecidas en el dominio. Esto implicó la inicialización de todas las funciones en el problema, aunque más tarde se tuvo que revertir esta acción. Algunas de estas funciones inicializadas se pueden apreciar en la Figura 3.8, donde se muestran las cargas iniciales para cada taxi y autobús definido. Además de eso, se eliminaron todos los tiempos asignados a cada acción, así como las distancias entre cada una de las localizaciones definidas.

```
(= (charge taxi1) 10)
(= (charge taxi2) 8)
(= (charge taxi3) 25)
(= (charge taxi4) 33)

(= (charge bus1) 20)
(= (charge bus1) 15)
```

Figura 3.8: Funciones que representan la carga inicial de taxis y autobuses

Finalmente, la última parte que necesitó ajustes en el problema fue la métrica a minimizar. Esto se debió a la carencia de acciones durativas y funciones en nuestro problema modificado. La métrica a minimizar original hacía uso de estas dos características que no estaban presentes en la versión adaptada del problema.

La función que representaba la métrica a minimizar se encuentra detallada en la Figura 3.9

```
(:metric minimize (+ (* 0.8 (total-time))  (* 0.2 (all-recharge-cost))))
)
```

Figura 3.9: Métrica a minimizar en el primer problema propuesto

Las demás características de un problema PDDL, detalladas en el apartado 2.2.2.1, que no se han abordado aquí, se han mantenido sin modificaciones y se explicarán en detalle en la sección siguiente.

3.5 Problema empleado

Como se ha comentado en el apartado 3.4 éste es el problema obtenido al modificar el expuesto en el mismo. Siguiendo el esquema expuesto en el apartado 2.2.2.1 comenzaremos explicando los objetos creados, los cuales podemos observarlos definidos en la Figura 3.10. Se han creado un total de cuatro objetos del tipo *taxi*, nombrados como *taxi1*, *taxi2*, *taxi3* y *taxi4*, dos del tipo *bus* nombrados como *bus1* y *bus2*, nueve objetos distintos del tipo *location* llamados *loc* seguidos de un número entre uno y nueve para diferenciarlos. Por último, seis objetos de tipo *passenger* definidos como *passenger* seguido de un número entre uno y seis.

```
(:objects
  taxi1 taxi2 taxi3 taxi4 - taxi
  bus1 bus2 - bus
  loc1 loc2 loc3 loc4 loc5 loc6 loc7 loc8 loc9 - location
  passenger1 passenger2 passenger3 passenger4 passenger5 passenger6 - passenger
)
```

Figura 3.10: Objetos del problema

Continuando con el esquema, tenemos el estado inicial, el cual se ha dividido en tres partes para que se pueda entender de manera más sencilla y ordenada. La primera parte se centra en la posición inicial de los objetos junto con la definición de que al inicio todos los taxis se encuentran vacíos, para la segunda parte tenemos todos los predicados para la unión de las localizaciones para los taxis, junto con un grafo para tener una imagen que lo defina visualmente. Para finalizar, tenemos todos los predicados de la unión de las rutas de los buses.

La primera parte la podemos observar en la Figura 3.11. Podemos observar que está separado por bloques: el primer bloque hace referencia a la localización inicial de cada taxi, por lo que por ejemplo *(at_taxi taxi1 loc1)* hace referencia a que el taxi 1 se encuentra en la localización 1 y así sucesivamente; el segundo bloque es donde se especifica que en nuestro caso todos los taxis se encuentran vacíos inicialmente con el predicado *(empty_taxi)* seguido de todos los objetos taxi; el tercer bloque es el que coloca a los buses en su posición inicial, de igual manera que el primer bloque lo hacia con los taxis, el predicado *(at_bus bus2 loc2)* coloca el segundo bus en la localización dos; en el cuarto y último bloque de esta parte tenemos la localización inicial para cada pasajero colocando por ejemplo al pasajero cinco y al seis en la localización nueve con los predicados *(passenger_at passenger5 loc9)* y *(passenger_at passenger6 loc9)*.

La segunda parte del estado inicial la podemos observar en la Figura 3.12. En este caso contamos con más bloques para que se pueda distinguir con mayor claridad qué localizaciones están conectadas entre sí, aunque se puede observar cada nodo representando una localización del problema de manera gráfica en la Figura 3.13. Cada bloque del código está representando una arista del grafo, ya que debemos conectar las dos localizaciones tanto en un sentido como en otro.

```
(:init

  (at_taxi taxi1 loc1)
  (at_taxi taxi2 loc2)
  (at_taxi taxi3 loc3)
  (at_taxi taxi4 loc5)

  (empty_taxi taxi1)
  (empty_taxi taxi2)
  (empty_taxi taxi3)
  (empty_taxi taxi4)

  (at_bus bus1 loc4)
  (at_bus bus2 loc2)

  (passenger_at passenger1 loc4)
  (passenger_at passenger2 loc6)
  (passenger_at passenger3 loc7)
  (passenger_at passenger4 loc8)
  (passenger_at passenger5 loc9)
  (passenger_at passenger6 loc9)
```

Figura 3.11: Primera parte del estado inicial del problema

```
(connected loc1 loc2)
(connected loc2 loc1)

(connected loc1 loc3)
(connected loc3 loc1)

(connected loc1 loc4)
(connected loc4 loc1)

(connected loc2 loc8)
(connected loc8 loc2)

(connected loc3 loc7)
(connected loc7 loc3)

(connected loc4 loc5)
(connected loc5 loc4)

(connected loc5 loc6)
(connected loc6 loc5)

(connected loc6 loc8)
(connected loc8 loc6)

(connected loc7 loc9)
(connected loc9 loc7)

(connected loc8 loc9)
(connected loc9 loc8)
```

Figura 3.12: Segunda parte del estado inicial del problema

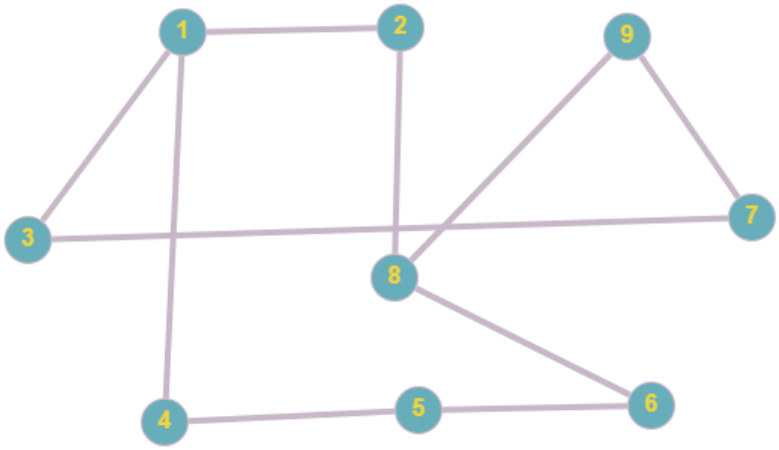

Figura 3.13: Grafo para representar la unión de localizaciones del problema

La tercera y última parte del estado inicial se centra en las rutas de los buses. Para ello se ha definido que los buses puedan acceder a cualquier localización desde cualquier punto. El código expuesto en la Figura 3.14 se ha simplificado para únicamente mostrar las conexiones de las tres primeras localizaciones, separadas también en bloques para una correcta visualización.

```
(route loc1 loc2)
(route loc1 loc3)
(route loc1 loc4)
(route loc1 loc5)
(route loc1 loc6)
(route loc1 loc7)
(route loc1 loc8)
(route loc1 loc9)

(route loc2 loc1)
(route loc2 loc3)
(route loc2 loc4)
(route loc2 loc5)
(route loc2 loc6)
(route loc2 loc7)
(route loc2 loc8)
(route loc2 loc9)

(route loc3 loc1)
(route loc3 loc2)
(route loc3 loc4)
(route loc3 loc5)
(route loc3 loc6)
(route loc3 loc7)
(route loc3 loc8)
(route loc3 loc9)
```

Figura 3.14: Tercera parte del estado inicial del problema

La parte final del esquema son los objetivos del problema, en nuestro caso hemos optado por definir una localización final para cada pasajero que sea distinta a su posición inicial, tomando otra localización de manera arbitraria, para el caso del pasajero 1 su localización objetivo sería la localización siete, para el pasajero dos sería la localización ocho, para el pasajero tres sería la localización nueve, para el pasajero cuatro sería la localización tres, para el pasajero cinco sería la localización cinco y para el pasajero seis sería la localización 1, como podemos observar en la Figura 3.15

```
(:goal (and
            (passenger_at passenger1 loc7)
            (passenger_at passenger2 loc8)
            (passenger_at passenger3 loc9)
            (passenger_at passenger4 loc3)
            (passenger_at passenger5 loc5)
            (passenger_at passenger6 loc1)
        ))
)
```

Figura 3.15: Objetivos del problema

3.6 Plan

Una vez definido nuestro dominio y problema en PDDL, llega el momento de obtener un plan mediante un planificador. Como se mencionó en la sección 2.2.3, hay una variedad de planificadores disponibles, pero en este trabajo nos centraremos en el planificador LPG. Este es el elegido por su uso en la asignatura de Técnicas, Entornos y Aplicaciones de Inteligencia Artificial, aunque cualquier otro de los mencionados podría ser utilizado con igual eficacia.

Para emplear el planificador LPG, se requiere ejecutarlo desde una línea de comandos junto con el archivo del dominio, el problema y los parámetros pertinentes. Por ejemplo:

```
1 .\lpg-td.exe -o .\Dominio.pddl -f .\Problema.pddl -n 3
```

En este caso, se especifica la ubicación del archivo del dominio con el parámetro '-o', la del archivo del problema con '-f' y el parámetro '-n 3' solicita tres soluciones, cada una mejor que la anterior.

Los planes generados pueden variar en longitud, como se muestra en la Figura 3.13, donde se observa un plan de treinta y tres acciones. Sin embargo, la cantidad de acciones puede diferir entre planes. Lo notable es que el plan devuelto utiliza todas las acciones definidas en el dominio y garantiza la consecución de los objetivos establecidos, ya que LPG siempre proporciona un plan válido.

Cada acción en el plan tiene un número que representa el instante de ejecución. Si varias acciones tienen el mismo número, indican que se ejecutan simultáneamente. Por ejemplo, las cuatro primeras acciones en el plan podrían estar ejecutándose al mismo tiempo.

```
0:   (DRIVE_TAXI TAXI1 LOC1 LOC4) [1]
0:   (DRIVE_BUS BUS1 LOC4 LOC9) [1]
0:   (DRIVE_TAXI TAXI3 LOC3 LOC7) [1]
0:   (DRIVE_BUS BUS2 LOC2 LOC6) [1]
0:   (DRIVE_TAXI TAXI4 LOC5 LOC6) [1]
1:   (PICKUP_TAXI TAXI1 PASSENGER1 LOC4) [1]
1:   (PICKUP_BUS BUS1 PASSENGER6 LOC9) [1]
1:   (PICKUP_TAXI TAXI3 PASSENGER3 LOC7) [1]
1:   (PICKUP_BUS BUS2 PASSENGER2 LOC6) [1]
1:   (DRIVE_TAXI TAXI4 LOC6 LOC8) [1]
2:   (DRIVE_TAXI TAXI1 LOC4 LOC1) [1]
2:   (DRIVE_BUS BUS1 LOC9 LOC1) [1]
2:   (DRIVE_TAXI TAXI3 LOC7 LOC9) [1]
2:   (DRIVE_BUS BUS2 LOC6 LOC8) [1]
2:   (DRIVE_TAXI TAXI4 LOC8 LOC9) [1]
3:   (DRIVE_TAXI TAXI1 LOC1 LOC3) [1]
3:   (DROPOFF_BUS BUS1 PASSENGER6 LOC1) [1]
3:   (DROPOFF_TAXI TAXI3 PASSENGER3 LOC9) [1]
3:   (DROPOFF_BUS BUS2 PASSENGER2 LOC8) [1]
3:   (PICKUP_TAXI TAXI4 PASSENGER5 LOC9) [1]
4:   (DRIVE_TAXI TAXI1 LOC3 LOC7) [1]
4:   (DRIVE_TAXI TAXI4 LOC9 LOC8) [1]
5:   (DROPOFF_TAXI TAXI1 PASSENGER1 LOC7) [1]
5:   (DRIVE_TAXI TAXI4 LOC8 LOC6) [1]
6:   (DRIVE_TAXI TAXI1 LOC7 LOC9) [1]
6:   (DRIVE_TAXI TAXI4 LOC6 LOC5) [1]
7:   (DRIVE_TAXI TAXI1 LOC9 LOC8) [1]
7:   (DROPOFF_TAXI TAXI4 PASSENGER5 LOC5) [1]
8:   (PICKUP_TAXI TAXI1 PASSENGER4 LOC8) [1]
9:   (DRIVE_TAXI TAXI1 LOC8 LOC9) [1]
10:   (DRIVE_TAXI TAXI1 LOC9 LOC7) [1]
11:   (DRIVE_TAXI TAXI1 LOC7 LOC3) [1]
12:   (DROPOFF_TAXI TAXI1 PASSENGER4 LOC3) [1]
```

Figura 3.16: Plan obtenido al ejecutar LPG

3.7 Validación, adaptación y reparación

Como se comento en la introducción del capitulo, este proceso de validación, adaptación y reparación se ha diseñado para ser genérico y que pueda funcionar con cualquier dominio, problema y plan definido siempre y cuando se mantengan dentro de la planificación clásica, usaremos el dominio y problema definidos anteriormente para que el proceso sea más sencillo de entender y nos ayude a proporcionar un ejemplo. Con la obtención de un dominio, un problema y un plan podemos empezar con la validación de éste. Para validar un plan se ha optado por usar la herramienta de validación VAL, para usarlo es necesario ejecutar la herramienta por consola, pasando las ubicaciones de los archivos del dominio, problema y plan como parámetros, de la siguiente forma:

```
1 .\Validate.exe -v -a .\Dominio.pddl .\Problema.pddl .\plan_Problema2.
    pddl_2.SOL
```

Al utilizar VAL con los argumentos específicos *-v* y *-a*, obtenemos un análisis detallado de nuestro plan, mostrado en la consola. Sin embargo, esta visualización no incluye las recomendaciones de reparación que VAL normalmente proporciona. Nuestro objetivo, una vez obtenido este análisis, es determinar si se puede reparar el plan para nuestro dominio y problema o replanificar es una mejor opción.

Cuando el plan resulta inválido, nuestro primer paso es identificar la acción específica en la que el plan deja de ser efectivo. Para ello, simulamos la ejecución del plan y observamos qué acción no se puede llevar a cabo o genera un resultado inesperado.

Posteriormente, retrocedemos al estado justo antes de que el plan dejara de ser válido y creamos un grafo de planificación simplificado. Este gráfico nos ayuda a calcular el coste de volver a cumplir las precondiciones necesarias para ejecutar la acción que falló. Esta información es esencial para comprender qué condiciones o acciones previas debemos restablecer para que la acción problemática funcione correctamente.

Sin embargo, corregir solo esa acción puede no ser siempre la mejor solución. Por este motivo, evaluamos la opción de unir otras acciones alrededor de la acción problemática. Buscamos acciones posteriores que nos acerquen a nuestros objetivos, considerando el costo de incorporar estas acciones adicionales al plan. Esto nos permite ofrecer al usuario una variedad de opciones para resolver el problema identificado.

Cuando presentamos estas alternativas al usuario, proporcionamos detalles sobre las acciones alternativas disponibles, sus costes respectivos y los objetivos que podrían alcanzarse al agregarlas al plan. Esto permite que el usuario tome una decisión informada sobre cómo proceder: unir las acciones sugeridas, mantener el plan sin cambios o buscar otras alternativas.

Si el usuario decide incorporar alguna de las acciones alternativas al plan, la ejecución continúa a partir de la acción a la que ha elegido unirse. En caso de que surjan más problemas con otras acciones, repetimos el proceso de análisis y ofrecemos nuevas opciones de corrección.

Tras todos estos intentos de reparación, si los objetivos deseados aún no se cumplen, ofrecemos la opción de realizar una replanificación completa. Esto implica volver a ejecutar el planificador LPG, proporcionándole el dominio y el problema, para obtener un nuevo plan que garantice ser válido y cumplir con todos los objetivos iniciales.

En la Figura 3.17 podemos observar un diagrama de flujo para que se pueda ver gráficamente el proceso a seguir para la validación y su posterior reparación o replanificación de un plan.

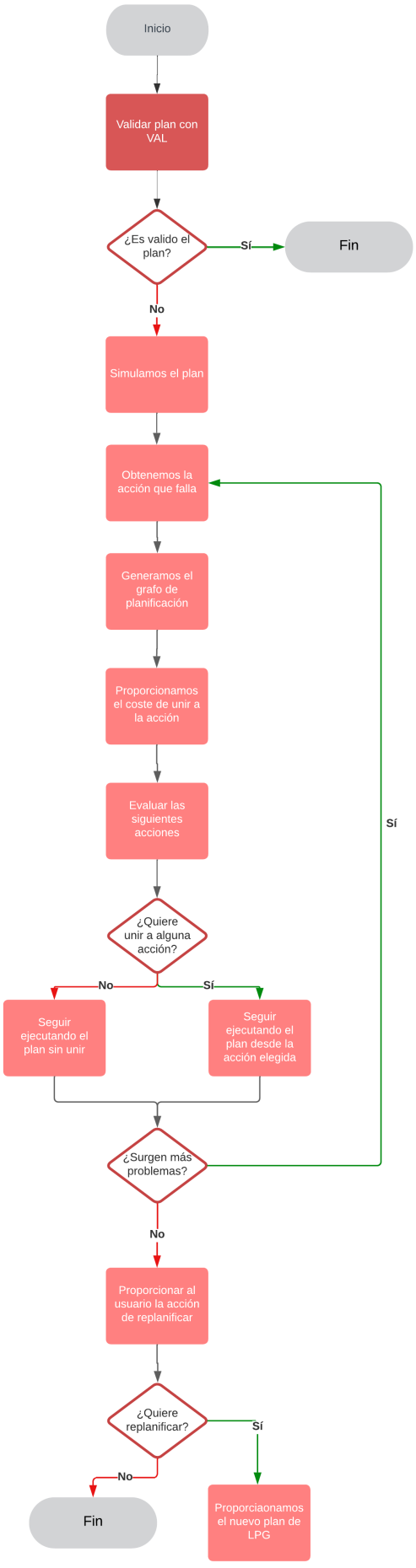

Figura 3.17: Esquema sobre el proceso a seguir

3.7.1. Ejemplo simplificado

Para el ejemplo simplificado vamos a tomar en cuenta el dominio y el problema definidos en las secciones 3.3 y 3.5 respectivamente, aunque cambiaremos los objetivos del problema, y un plan reducido del generado por LPG, el plan que estaremos analizando se muestra en la Figura 3.18, que como se puede observar cuenta únicamente con nueve acciones. Los nuevos objetivos a cumplir para nuestro problema serán *'(passenger_at passenger1 loc7)'* y *'(on_board_bus passenger2 bus2)'*.

```
0:   (PICKUP_BUS BUS1 PASSENGER1 LOC4) [1]
0:   (DRIVE_TAXI TAXI2 LOC2 LOC8) [1]
0:   (DRIVE_TAXI TAXI3 LOC3 LOC7) [1]
1:   (PICKUP_TAXI TAXI2 PASSENGER4 LOC8) [1]
1:   (PICKUP_TAXI TAXI3 PASSENGER3 LOC7) [1]
1:   (PICKUP_BUS BUS2 PASSENGER2 LOC6) [1]
1:   (DRIVE_BUS BUS1 LOC4 LOC7) [1]
2:   (DROPOFF_BUS BUS1 PASSENGER1 LOC7) [1]
3:   (DROPOFF_TAXI TAXI3 PASSENGER3 LOC9) [1]
```

Figura 3.18: Plan reducido

Una vez se obtiene el nuevo plan, el primer paso es pasar por la herramienta VAL, como se ha explicado previamente. Con el plan reducido, VAL indica que la acción *'(pickup_bus bus2 passenger2 loc6)'* no puede llevarse a cabo. Comenzamos entonces a intentar reparar esta acción, iniciando con la simulación del plan. Esto implica ejecutar las acciones desde el inicio hasta justo antes de la acción identificada como errónea.

Al llegar a este punto en la secuencia de acciones, generamos un grafo de planificación relajado para calcular el costo de reintegrar esa acción al plan. En este caso, el costo de reintegrar la acción desde nuestro estado actual es de 2. Para lograr nuevamente todos los objetivos propuestos, debemos asumir este costo. Alternativamente, podríamos unir alguna de las acciones posteriores, donde el costo de unión desde nuestro estado actual sería de 0 para las acciones *'(drive_bus bus1 loc4 loc7)'* y *'(dropoff_bus bus1 passenger1 loc7)'* y de 3 para la acción *'(dropoff_taxi taxi3 passenger3 loc9)'*.

Si optamos por no asumir el coste de 2 para la acción rota y elegimos unir una acción con coste 0 o 3, no lograríamos alcanzar todos los objetivos, ya que la acción rota es necesaria para otro objetivo pendiente. Por lo tanto, el usuario debe decidir qué compensa más: asumir el costo extra, unir una acción pero no alcanzar todos los objetivos o continuar sin unir nada.

En este ejemplo, se muestran todas las opciones posibles. En este caso, el usuario decide unir la acción *'(dropoff_bus bus1 passenger1 loc7)'* y continuar con el plan, aunque no se alcancen todos los objetivos. Sin embargo, al llegar a la última acción, se descubre que también es incorrecta, ya que el taxi3 no se encuentra en la localización 9, impidiendo dejar al pasajero en esa ubicación. Dado este hallazgo, el proceso vuelve a empezar.

En este caso, al ser la última acción del plan, solo se devuelve el costo para unir esta acción, que ahora es de 2 debido a que nos encontramos en un estado

distinto al momento del primer intento de unión a esta acción. Aunque asumir este costo solo nos permitiría alcanzar uno de los dos objetivos, el usuario decide no unir la acción.

En este punto, se le pregunta al usuario si prefiere una replanificación completa para lograr todos los objetivos o quedarse con el plan reparado. Como queremos explorar todas las opciones, el usuario elige la replanificación completa, obteniendo así un nuevo plan válido que alcanza todos los objetivos que podemos observar en la Figura 3.19.

```
0:   (PICKUP_BUS BUS1 PASSENGER1 LOC4) [1]
0:   (DRIVE_TAXI TAXI2 LOC2 LOC8) [1]
0:   (DRIVE_TAXI TAXI3 LOC3 LOC7) [1]
0:   (DRIVE_BUS BUS2 LOC2 LOC6) [1]
1:   (PICKUP_TAXI TAXI2 PASSENGER4 LOC8) [1]
1:   (PICKUP_TAXI TAXI3 PASSENGER3 LOC7) [1]
1:   (PICKUP_BUS BUS2 PASSENGER2 LOC6) [1]
1:   (DRIVE_BUS BUS1 LOC4 LOC7) [1]
2:   (DROPOFF_BUS BUS1 PASSENGER1 LOC7) [1]
```

Figura 3.19: Plan reducido correcto

CAPÍTULO 4
Implementación

Con esta sección se quiere entrar más en detalle sobre la implementación y las tecnologías utilizadas para llevar acabo el proyecto, mostrando además las pruebas realizadas para comprobar el código.

4.1 Herramientas y tecnologías

Para este proyecto, se ha empleado Python como lenguaje principal de programación junto con Visual Studio Code como entorno de desarrollo. Se han utilizado diversas librerías de Python para mejorar la eficiencia y funcionalidad en la implementación del proyecto.

- **Python:** Python es un lenguaje de programación interpretado y de alto nivel que prioriza la legibilidad del código. Su filosofía subraya la claridad y la simplicidad. Es un lenguaje multiparadigma, admitiendo la orientación a objetos, la programación imperativa y, en menor medida, la programación funcional. Es interpretado, dinámico y compatible con múltiples plataformas[14]. Se ha utilizado en el proyecto por la familiaridad con él, así como por sus bibliotecas relacionadas con la planificación y su flexibilidad y versatilidad permitiendo adaptarse a diferentes enfoques de programación.

- **Visual Studio Code:** Visual Studio Code, comúnmente conocido como VS Code, es un editor de código desarrollado por Microsoft compatible con Windows, Linux, macOS y la web. Ofrece una amplia gama de funciones, que incluyen soporte para depuración, control integrado de Git, resaltado de sintaxis, completado inteligente de código, manejo de fragmentos y refactorización de código. En este caso se ha usado por la amplia cantidad de extensiones que nos permiten ver y modificar con claridad todos los distintos tipos de archivos del proyecto, desde el dominio y problema en PDDL hasta el plan obtenido por LPG.

- **Librerías:** Una librería es un conjunto de funciones, clases y recursos que se pueden utilizar en el desarrollo de software. Estas librerías suelen ser creadas por otros desarrolladores y se ponen a disposición del público para su uso[15]. Para nuestro proyecto se han usado dos librerías principales:

1. **Aima-python**[1]: Es una librería que implementa todo el pseudocódigo del libro Artificial Intelligence: A Modern Approach, al lenguaje de programación Python, en nuestro caso nos centramos únicamente en el pseudocódigo descrito en los capítulos 10 y 11 del libro, centrados en la planificación. Esta librería nos ayuda con la creación del grafo de planificación y con el uso de acciones para la simulación del plan.
2. **Pddlpy**[2]: Esta librería es un *parser* que traduce un archivo PDDL para que podamos interactuar con él en Python, por lo que nos es increíblemente útil a la hora de pasar nuestro dominio y problema a Python y poder trabajar con ellos de manera simple.

4.2 Programa desarrollado

El programa en cuestión es una herramienta diseñada para validar planes y repararlos en caso de que sea necesario con ayuda del usuario. Para comenzar, el usuario debe proporcionar un dominio, un problema y un plan. Los dos primeros elementos son analizados utilizando la librería *pddlpy*. Sin embargo, surge un pequeño obstáculo: el formato en el que se obtienen estos elementos no es directamente compatible con la librería *Aima-python*, por lo que se requiere una adaptación.

Para resolver este problema de compatibilidad, se realiza una conversión o ajuste en el formato de los elementos del dominio y el problema. Este proceso de adaptación es necesario para poder trabajar efectivamente con la librería *Aima-python*. Una vez lograda la compatibilidad, se crea un objeto de la clase *PlanningProblem* que será utilizado posteriormente para generar un grafo de planificación y ejecutar las acciones en la simulación del plan.

El siguiente paso es la lectura y almacenamiento de las acciones del plan proporcionado. Estas acciones se guardan en un *array*, lo que facilita su manipulación y procesamiento posterior en Python.

Ahora, procedemos a utilizar VAL siguiendo el mismo proceso detallado en la sección 3.7. Si la validación revela que el plan es incorrecto, llevamos a cabo una simulación del plan para identificar la acción defectuosa. Todo este proceso se ejecuta a través de una función específica, representada en el algoritmo 4.1. Esta metodología nos permite obtener nuestro dominio y problema en Python para trabajar con ellos, además de realizar una validación inicial de nuestro plan. En caso de incongruencias, la simulación del plan nos ayuda a identificar la acción defectuosa.

Una vez determinamos un plan erróneo, procedemos con la simulación para identificar la acción que ha fallado. Aunque VAL nos proporciona información sobre la acción defectuosa y una posible corrección, en esta ocasión obviamos ese detalle para enfocarnos en la reparación directa. La simulación nos ofrece el estado del problema antes de que la acción fallara, lo que nos permite mejorar el plan desde ese punto y realizar ajustes más efectivos.

[1] https://github.com/aimacode/aima-python
[2] https://github.com/hfoffani/pddl-lib

Algorithm 4.1 checkPlan

```
Require: domainFile, problemFile, planFile, problem, accionesP
valPath ← 'RUTA_AL_ARCHIVO_VALIDATE.EXE'
result ← ejecutarValidación(valPath, domainFile, problemFile, planFile)
escribirResultadoEnArchivo(result)
líneas ← leerArchivo(result)
planValido ← Falso

for cada línea en líneas (de manera reversa) do
    planFallido ← verificarPlanFallidoEnLínea(línea)
    if 'Plan válido' está en línea then
        planValido ← Verdadero
        Detener el ciclo
    else if planFallido then
        planValido ← Falso
    end if
end for

if planValido then
    Mostrar('El plan es valido')
    simularPlanValido(accionesP, problem)
else
    Mostrar('El plan es inválido')
    simularPlanInvalido(accionesP, problem)
end if
```

La librería *Aima-python* ofrece un método que permite que una acción afecte al problema generado previamente, lo que posibilita simular el plan paso a paso. Este método puede lanzar una excepción si la acción que intentamos ejecutar no puede realizarse dentro del contexto actual del problema. Utilizando un bloque *try-catch* al invocar este método, identificamos qué acción ha fallado, lo que nos permite comenzar la reparación del plan en consecuencia desde el estado en el que nos encontramos.

Mientras simula el plan, el programa guarda las acciones ejecutadas con éxito en un nuevo *array*. Esto nos permite reconstruir el plan una vez que se hayan realizado las correcciones necesarias. Podemos observar el código de la función que simula el plan en el algoritmo 4.2.

Algorithm 4.2 simularPlanInvalido

Require: planActions, problem
failedActions ← ListaVacia
newPlan ← ListaVacia
brokenAction ← Falso
for cada acción en planActions **do**
  try
    ejecutarAcciónEnProblema(acción, problema)
    agregarAcciónAlNuevoPlan(acción, newPlan)
  except Excepción como e
    índice ← encontrarÍndiceAcción(acción, planActions)
    subArray ← obtenerSubarregloDesdeÍndice(índice, planActions)
    agregarAcciónFallida(acción, failedActions)
    planActions ← adjuntarPlan(subArray, problema)
end for
metasAlcanzadas ← ObtenerMetasAlcanzadas(problema)
Mostrar('Nuevo plan reparado:', newPlan)
Mostrar('Acciones que han fallado:', failedActions)
Mostrar('Objetivos totales conseguidos:', metasAlcanzadas)

Dentro del contexto de la función referenciada en 4.2, se encuentra una función que se encarga de calcular y devolver el plan restante al elegir qué acción unir. Esta función presenta al usuario todas las posibles acciones que pueden unirse, asegurando que al menos lleguen a un objetivo. Además, muestra el coste obtenido del grafo de planificación relajado para unir cada acción.

El proceso inicia al proporcionar a este método un *array* que contiene las acciones pendientes por ejecutarse, incluyendo la acción que ha fallado. Para cada una de estas acciones, se procede a calcular primero el costo de unir esa acción desde el estado actual del problema, justo antes de que la acción falle, utilizando el grafo de planificación relajado. Posteriormente, se simula el plan con las acciones restantes que aún deben realizarse.

Dado que este proceso se realiza para cada acción, el plan restante será siempre más reducido. El propósito de la simulación es obtener información sobre los objetivos que se alcanzarán al unir cada acción. Esta información facilita al usuario la elección de la acción más conveniente.

El fragmento de código que ilustra este procedimiento se encuentra detallado en 4.3. Una vez que se ha recopilado toda esta información, se solicita al usuario que elija el número de la acción que desea unir. Alternativamente, puede optar por continuar con el plan sin unir ninguna acción, escribiendo el número 0. Dependiendo de la acción seleccionada por el usuario, se determina su índice correspondiente y se devuelve el *array* restante, que comprende desde la acción elegida hasta el final del plan. Se puede observar en el fragmento del código descrito en 4.4.

En caso de que una acción vuelva a fallar dentro del nuevo plan devuelto a la función 4.5, se llamará nuevamente a esta función con lo que se vuelve a realizar el proceso de reparación del plan.

Este enfoque de análisis y selección de acciones para unir se apoya en la iteración continua del proceso de planificación, brindando al usuario la capacidad de tomar decisiones informadas y ajustar el plan en función de las circunstancias emergentes.

Algorithm 4.3 Primera parte attachPlan

Require: subAcciones, problema

```
num_accion ← 1
for cada acción principal en subAcciones do
    nuevo_problema ← CrearNuevoProblema(problema)
    nombreAccion ← ObtenerNombreAccion(accion_principal)
    args ← ObtenerArgumentos(accion_principal)
    nuevosObjetivos ← ListaVacia
    accionesRelajadas ← ListaVacia
    accionesFallidas ← ListaVacia

    for cada acción en nuevo_problema.acciones do
        accionesRelajadas.agregar(relajarAcción(acción))
        if nombreAccion está en acción.nombre then
            for all cláusula en acción.precond do
                nuevosObjetivos.agregar(sustituirObjetivo(cláusula, args))
            end for
        end if
    end for

    problemaRelajado ← CrearProblemaRelajado(nuevo_problema, nuevo-
    sObjetivos, accionesRelajadas)
    num_nivel ← 0
    grafoRelajado ← GraficarPlanRelajado(problemaRelajado)
    estado_actual ← grafoRelajado.obtenerEstadoActual()
    objetivos_conseguidos ← ConvertirAConjunto(nuevosObjetivos)

    while no están todos los objetivos_conseguidos en estado_actual do
        grafoRelajado.expandirGrafo()
        num_nivel ← num_nivel + 1
        estado_actual ← grafoRelajado.obtenerEstadoEnNivel(num_nivel)
    end while

    índice ← EncontrarÍndice(accion_principal, subAcciones)
    subArray ← ObtenerSubArrayDesdeÍndice(índice, subAcciones)

    for cada acción en subArray do
        try:
            nuevo_problema.act(ejecutarExpresión(accion))
            accionesRealizadas.agregar(accion)
        except Excepción como e:
            accionesFallidas.agregar(accion)
    end for

    Cerramos el bucle FOR para que latex nos deje continuar el algoritmo en
    otra pagina ya que de otra forma no se podría

end for
```

Algorithm 4.4 Segunda parte attachPlan

```
(Es el mismo bucle que en la parte 1, continua hasta que este bucle se cierra)

for cada acción principal en subAcciones do
    objetivosConseguidos ← ObtenerObjetivosConseguidos(nuevo_problema)

    if no hay objetivosConseguidos then
        detener el bucle
    end if

    Mostrar('Acción a unir:', accion_principal)
    Mostrar('Número de acción:', num_accion)
    Mostrar('Niveles expandidos en el plan relajado:', num_nivel)
    Mostrar('Acciones que no se realizaran al unir la acción:', accionesFallidas)

    Mostrar('Objetivos conseguidos si se une la acción:', objetivosConseguidos)
    num_accion ← num_accion + 1
end for

try:
    numero ← ObtenerEntradaUsuario(Ïngrese el número de la acción que desea unir. Ingrese 0 si no desea unir ninguna acción:")
except ValueError
    Mostrar("Por favor, introduce un número válido")
    Salir()


if numero es igual a 0 then
    return subAcciones[1:]
else
    posición ← numero - 1
    return subAcciones[posición:]
end if
```

4.2.1. Pruebas realizadas

Durante las pruebas realizadas para la ejecución total del plan, el enfoque se centró en verificar el recorrido del código, utilizando salidas de texto por consola para asegurar su ejecución correcta.

Se implementaron dos enfoques distintos. El primero implicó la modificación manual del plan generado por LPG, omitiendo deliberadamente alguna de sus acciones. En el segundo método, se introdujo una variable de probabilidad para evitar la ejecución de una acción específica. Estos enfoques permitieron realizar pruebas en situaciones donde el plan se invalidaba desde el inicio, así como en

escenarios donde un plan válido se interrumpía al omitir una acción aleatoria, lo que ocasionaba la interrupción del programa. A continuación, se presentarán ejemplos de las ejecuciones obtenidas al emplear estos métodos diferentes.

Con el primer método, se espera que el plan sea inválido desde el inicio. Para verificar esto, hemos incluido una línea en el código que imprime en la consola si el resultado de VAL indica que el plan es inválido. Una vez que llegamos a este punto, ejecutamos el algoritmo para localizar la acción que está causando el problema. Utilizamos un bloque *try-catch*, como se muestra en el código 4.5, para identificar y manejar el error cuando una acción falla.

Cuando se detecta la acción problemática y se empieza a corregir el plan, priorizamos la unión de la primera acción que provoca el fallo. Mostramos toda la información relacionada con esa acción. En la Figura 4.1 se puede observar que se imprime la línea 'El plan es inválido', con lo que sabemos que el código esta siguiendo la traza esperada, en la figura se muestra también la primera acción a unir junto con toda su información. Luego, presentamos las otras acciones del plan con las que se puede unir, mostrando la información correspondiente. Finalmente, le preguntamos al usuario a qué número de acción quiere unirse, y el resultado se imprime por consola como se muestra en la Figura 4.2.

```
El plan es inválido
Accion a unir: Drive_taxi(TAXI4, LOC9, LOC8)
Acción número: 1
Niveles expandidos en el graph plan relajado: 0
Acciones que no se realizaran al unir la acción: ['Dropoff_taxi(TAXI1, PASSENGER1, LOC7)', 'Drive_taxi(TAXI1, LOC7, LOC9)', 'Drive_taxi(TAXI1, LOC9,
LOC8)', 'Pickup_taxi(TAXI1, PASSENGER4, LOC8)', 'Drive_taxi(TAXI1, LOC8, LOC9)', 'Drive_taxi(TAXI1, LOC9, LOC7)', 'Drive_taxi(TAXI1, LOC7, LOC3)', 'D
ropoff_taxi(TAXI1, PASSENGER4, LOC3)']
Objtivos conseguidos si unes la acción: [Passenger_at(PASSENGER6, LOC1), Passenger_at(PASSENGER3, LOC9), Passenger_at(PASSENGER2, LOC8), Passenger_at
(PASSENGER5, LOC5)]
```

Figura 4.1: Primera parte de la ejecución

```
Accion a unir: Drive_taxi(TAXI1, LOC7, LOC3)
Acción número: 11
Niveles expandidos en el graph plan relajado: 2
Acciones que no se realizaran al unir la acción: ['Drive_taxi(TAXI1, LOC7, LOC3)', 'Dropoff_taxi(TAXI1, PASSENGER4, LOC3)']
Objtivos conseguidos si unes la acción: [Passenger_at(PASSENGER6, LOC1), Passenger_at(PASSENGER3, LOC9), Passenger_at(PASSENGER2, LOC8)]

Accion a unir: Dropoff_taxi(TAXI1, PASSENGER4, LOC3)
Acción número: 12
Niveles expandidos en el graph plan relajado: 3
Acciones que no se realizaran al unir la acción: ['Dropoff_taxi(TAXI1, PASSENGER4, LOC3)']
Objtivos conseguidos si unes la acción: [Passenger_at(PASSENGER6, LOC1), Passenger_at(PASSENGER3, LOC9), Passenger_at(PASSENGER2, LOC8)]

Ingrese el número con la acción quiere unir, ingrese el número 0 si no quiere unir ninguna de las acciones:
```

Figura 4.2: Últimas acciones a unir y pregunta al usuario

Una vez se identifica la acción a la que el usuario desea unirse, se imprime por pantalla para verificar que es la acción esperada. Continuamos siguiendo la secuencia esperada del plan hasta encontrar la siguiente acción que esté defectuosa. Este proceso de corrección se repite de manera similar a los ejemplos previamente mencionados.

Al completar la ejecución del plan y unir todas las acciones necesarias, presentamos al usuario el plan resultante de la reparación. Este incluye tanto las acciones que han presentado fallos como los objetivos finales alcanzados. Además,

al finalizar, se plantea al usuario la última pregunta sobre si desea una replanificación completa, como se muestra en la Figura 4.3

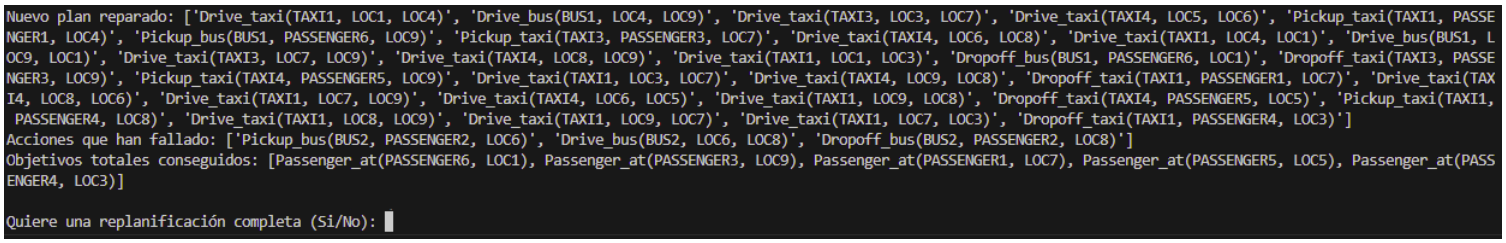

Figura 4.3: Plan obtenido junto con las acciones que no se han realizado y los objetivos obtenidos

En el segundo método, se espera que el plan sea válido desde el inicio, incluyendo una línea similar al primer método para verificar que el plan sigue la secuencia esperada. La línea que se imprime para este caso es 'El plan es válido'. Sin embargo, para incluir pruebas incluso cuando el plan es válido, se ha adaptado el algoritmo mostrado en 4.2. Esta adaptación incorpora la implementación de una variable global que determina la probabilidad de que una acción falle, establecida en un 7 %. Se limita la cantidad de acciones que pueden fallar con este método a una, dado que la falla de una acción garantiza el fallo de las acciones siguientes, haciendo innecesario provocar más de un error con este método. Podemos observar la modificación de la función en 4.5.

Algorithm 4.5 simularPlanValido

```
Require: planActions, problem
brokenAction ← Falso
for cada accion en planActions do
    randomNumber ← generarNumeroAleatorio()
    if (randomNumber ≤ porcentajeFalloAccion) y no brokenAction then
        agregarAccionNoRealizada(accion, unperformedActions)
        index ← encontrarÍndiceAccion(accion, planActions)
        subArray ← obtenerSubArrayDesdeÍndice(index, planActions)
        brokenAction ← Verdadero
        planActions ← adjuntarPlan(subArray, problem)
    else
        try:
            ejecutarAccionEnProblema(accion, problem)
            agregarAccionAlNuevoPlan(accion, newPlan)
        except Excepción como e:
            index ← encontrarÍndiceAccion(accion, planActions)
            subArray ← obtenerSubArrayDesdeÍndice(index, planActions)
            agregarAccionFallida(accion, failedActions)
            planActions ← adjuntarPlan(subArray, problem)
    end if
end for
```

La primera acción defectuosa se señala con la línea 'Acción rota', tal como se muestra en la Figura 4.4. Posteriormente, para las acciones subsecuentes afec-

tadas por esta primera falla, se utiliza la línea 'Acción ha fallado'. Este proceso permite seguir la traza del código, asegurándonos de que se sigue la secuencia correcta, como se ilustra en la Figura 4.5. La distinción entre la primera acción rota y las consecuentes fallas derivadas de esta proporciona una visualización clara de las acciones problemáticas en el plan.La secuencia de trazas finaliza de manera similar a cuando el plan es inválido desde el inicio. En este punto, se presenta al usuario la pregunta sobre si desea realizar una replanificación completa, tal como se mostraba en la Figura 4.3.

```
El plan es valido
Acción rota
Accion a unir: Drive_bus(BUS2, LOC2, LOC6)
Acción número: 1
Niveles expandidos en el graph plan relajado: 0
Acciones que no se realizaran al unir la acción: []
Objtivos conseguidos si unes la acción: [Passenger_at(PASSENGER6, LOC1), Passenger_at(PASSENGER3, LOC9), Passenger_at(PASSENGER2, LOC8), Passenger_at(PASSENGER1, LOC7), Passenge
r_at(PASSENGER5, LOC5), Passenger_at(PASSENGER4, LOC3)]
```

Figura 4.4: Líneas impresas por consola al romper la primera acción

```
Accion ha fallado
Accion a unir: Pickup_bus(BUS2, PASSENGER2, LOC6)
Acción número: 1
Niveles expandidos en el graph plan relajado: 1
Acciones que no se realizaran al unir la acción: ['Pickup_bus(BUS2, PASSENGER2, LOC6)', 'Drive_bus(BUS2, LOC6, LOC8)', 'Dropoff_bus(BUS2, PASSENGER2, LOC8)']
Objtivos conseguidos si unes la acción: [Passenger_at(PASSENGER6, LOC1), Passenger_at(PASSENGER3, LOC9), Passenger_at(PASSENGER1, LOC7), Passenger_at(PASSENGER5, LOC5), Passenge
r_at(PASSENGER4, LOC3)]
```

Figura 4.5: Líneas impresas por consola al obtener acciones fallidas

CAPÍTULO 5
Conclusión

El desarrollo de este trabajo ha sido fundamental para alcanzar los objetivos planteados desde su inicio. La creación de un modelo detallado en el ámbito de la movilidad urbana en *smart cities*, como se detalla en la sección 3.1, junto con la implementación efectiva de la herramienta de validación VAL y la aplicación de una heurística de relajación mediante un grafo de planificación en la sección 4.2, ha sido un paso significativo hacia la obtención de soluciones flexibles y adaptativas.

El análisis del proceso de reparación de planes defectuosos, proporcionando datos al usuario para la toma de decisiones, otorgando así la capacidad de adaptar los planes a las preferencias individuales del usuario o, en su defecto, generar una replanificación integral mediante el planificador LPG (conforme se detalla en la sección 3.4).

Sin embargo, es crucial reconocer las áreas que requieren una mayor exploración y expansión. Aunque se lograron los objetivos establecidos, la ausencia de pruebas con diferentes dominios y problemas es una limitación que merece atención. La introducción de estas pruebas podría enriquecer significativamente la comprensión del rendimiento del código en contextos diversos y, por ende, ampliar su aplicabilidad.

Además, la posibilidad de experimentar con una gama más amplia de planificadores que utilicen diversas técnicas y heurísticas podría arrojar luz sobre cómo estas variaciones podrían afectar los resultados.

En resumen, este trabajo ha cumplido con los objetivos establecidos y ha demostrado su eficacia en la resolución de problemas específicos dentro del dominio de la movilidad urbana en *smart cities*. No obstante, se presenta una oportunidad valiosa para una investigación más amplia y profunda que involucre la aplicación del código en una variedad de contextos y la evaluación de múltiples enfoques de planificación. Esta ampliación en la exploración podría enriquecer aún más la comprensión y aplicabilidad de los resultados obtenidos.

Este trabajo ha aprovechado los conocimientos que se han obtenido de cursas las diferentes asignaturas dentro de la rama de Computación:

- **Técnicas, Entornos y Aplicaciones de Inteligencia Artificial:** esta asignatura ha sido fundamental como base del proyecto, ya que proporcionó todos los fundamentos necesarios para abordar los temas principales del trabajo.

En ella se exploraron aspectos clave como la planificación, una variedad de técnicas heurísticas y el lenguaje PDDL, proporcionando los conocimientos fundamentales que sirvieron como punto de partida para el desarrollo de este proyecto.

- **Sistemas inteligentes:** representó un punto de partida crucial al introducir el concepto de Inteligencia Artificial, así como distintos algoritmos asociados.

- **Percepción:** se introdujo al alumno en el lenguaje de programación Python, el cual ha sido fundamental para el desarrollo de una parte significativa de este proyecto.

5.1 Trabajo futuro

A pesar de haber cumplido los objetivos establecidos, existen numerosas oportunidades de mejora y desarrollo que podrían enriquecer y completar el proyecto. Aunque se han mencionado algunas de estas mejoras de manera general en la conclusión, entre las áreas a explorar más detalladamente se encuentran:

- **Uso de otras técnicas heurísticas:** El proyecto se ha centrado en una sola heurística y un enfoque específico utilizando el grafo de planificación. Sin embargo, una mejora sustancial radicaría en la exploración y aplicación de diversas heurísticas, como las que se detallan en el apartado 2.2, así como en la adopción de múltiples enfoques para el método ya utilizado.

  La experimentación con un conjunto diverso de heurísticas y enfoques permitiría una comparación más exhaustiva de sus efectos en la calidad y eficiencia de los planes generados. Asimismo, esta exploración podría revelar patrones o estrategias más efectivas según el tipo de problema o dominio particular, abriendo puertas a la optimización y personalización de soluciones futuras.

- **Ampliación para el uso de dominios y problemas más complejos:** A lo largo del proyecto se ha trabajado con dominios y problemas de planificación clásica. No obstante, una mejora significativa radicaría en la adaptación del enfoque para abordar problemas más complejos. Como por ejemplo, el definido en 3.2.

  La extensión hacia problemas más complejos enriquecería la capacidad del enfoque utilizado, permitiendo su aplicación en escenarios más realistas y variados. La inclusión de costos en las acciones, la consideración del tiempo como factor crucial en la planificación y la incorporación de elementos numéricos proporcionarían un marco más robusto para abordar situaciones que reflejen de manera más fiel la complejidad del mundo real.

- **Creación de una heurística para ofrecer acciones al usuario:** Otra mejora potencial radica en la creación de una heurística para filtrar y presentar acciones al usuario. En lugar de mostrar todas las acciones que conduzcan al

menos a un objetivo, esta estrategia se centraría en presentar únicamente las acciones más relevantes y efectivas.

Esta optimización pretende ofrecer al usuario una experiencia más intuitiva y eficiente al reducir la cantidad de opciones de acciones disponibles. Al presentar únicamente las acciones más pertinentes y efectivas, se simplificaría el proceso de toma de decisiones, permitiendo una selección más rápida y consciente por parte del usuario.

Bibliografía

[1] GHALLAB, M., NAU, D.S. y TRAVERSO, P., 2016. Automated planning and acting. New York, NY: Cambridge University Press. ISBN 9781139583923.

[2] HOWEY, R., LONG, D. y FOX, M., 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. 16th IEEE International Conference on Tools with Artificial Intelligence [en línea]. Boca Raton, FL, USA: IEEE Comput. Soc, pp. 294-301. [consulta: 11 noviembre 2023]. ISBN 9780769522364. DOI 10.1109/ICTAI.2004.120. Disponible en: http://ieeexplore.ieee.org/document/1374201/.

[3] MULLER, P., FONTRODONA, J. 2020. Smart cities y ciudadanía inteligente: Tecnología, privacidad y desarrollo. Cátedra CaixaBank de Responsabilidad Social Corporativa. Disponible en: https://media.iese.edu/research/pdfs/ST-0606.pdf.

[4] BABLI, M., SAPENA, Ó. y ONAINDIA, E., 2023. Plan commitment: Replanning versus plan repair. Engineering Applications of Artificial Intelligence [en línea], vol. 123, [consulta: 11 noviembre 2023]. ISSN 09521976. DOI 10.1016/j.engappai.2023.106275. Disponible en: https://linkinghub.elsevier.com/retrieve/pii/S0952197623004591

[5] RUSSELL, S.J., NORVIG, P. y DAVIS, E., 2010. Artificial intelligence: a modern approach. 3rd ed. Upper Saddle River: Prentice Hall. Prentice Hall series in artificial intelligence, ISBN 9780136042594. Q335 .R86 2010

[6] Planning Domain Definition Language. En: Page Version ID: 1137801204, Wikipedia [en línea], 2023. [consulta: 11 noviembre 2023]. Disponible en: https://en.wikipedia.org/w/index.php?title=Planning_Domain_Definition_Language&oldid=1137801204.

[7] PDDL. Planning.wiki - The AI Planning & PDDL Wiki [en línea]. [consulta: 11 noviembre 2023]. Disponible en: https://planning.wiki/ref/pddl.

[8] The LPG planning system. [en línea]. [consulta: 13 noviembre 2023]. Disponible en: https://ipc02.icaps-conference.org/LPG.html.

[9] NEXUSADMISTRAINTEGRA, 2021. Smart City, cómo funcionan y cómo abordar el cambio. Nexus Integra [en línea]. [consulta: 16 noviembre 2023]. Disponible en: https://nexusintegra.io/es/smart-city-que-es-como-funciona/.

[10] MIPS-XXL Planner «Shahid Jabbar, Ph.D. [en línea]. [consulta: 17 noviembre 2023]. Disponible en: http://sjabbar.com/mips-xxl-planner.

[11] Planning at KCL. [en línea]. [consulta: 17 noviembre 2023]. Disponible en: https://nms.kcl.ac.uk/planning/software/optic.html.

[12] SGPlan. [en línea]. [consulta: 17 noviembre 2023]. Disponible en: https://wah.cse.cuhk.edu.hk/wah/programs/SGPlan/.

[13] HOFFMANN, J., PORTEOUS, J. y SEBASTIA, L., 2004. Ordered Landmarks in Planning. Journal of Artificial Intelligence Research [en línea], vol. 22, [consulta: 18 noviembre 2023]. ISSN 1076-9757. DOI 10.1613/jair.1492. Disponible en: https://jair.org/index.php/jair/article/view/10390.

[14] Python. En: Page Version ID: 155353048, Wikipedia, la enciclopedia libre [en línea], 2023. [consulta: 20 noviembre 2023]. Disponible en: https://es.wikipedia.org/w/index.php?title=Python&oldid=155353048.

[15] INÁBA, S., 2023. Significado de librería en programación. Soluciones Inába [en línea]. [consulta: 20 noviembre 2023]. Disponible en: https://www.inabaweb.com/significado-de-libreria-en-programacion/.

[16] FOX, M., GEREVINI, A., LONG, D. y SERINA, I., 2006. Plan Stability: Replanning versus Plan Repair. ICAPS 2006 - Proceedings, Sixteenth International Conference on Automated Planning and Scheduling. Disponible en: https://www.researchgate.net/publication/220936293_Plan_Stability_Replanning_versus_Plan_Repair

[17] ABEL, D., 2020. A Theory of Abstraction in Reinforcement Learning. Ph.D, Brown University. Disponible en: https://arxiv.org/pdf/2203.00397.pdf

APÉNDICE A
Objetivos de Desarrollo Sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (*ODS*).

Objetivos de Desarrollo Sostenible	Alto	Medio	Bajo	No procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**			X	
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**				X
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**		X		
ODS 12. **Producción y consumo responsables.**				
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG con los *ODS* y con los *ODS* más relacionados.

Los Objetivos de Desarrollo Sostenible (ODS) 7, 11 y 13 tienen una interrelación significativa en el contexto de las ciudades inteligentes o smart cities, donde la validación, reparación y adaptación de planes son fundamentales para alcanzar metas globales. Estos objetivos buscan abordar la energía asequible y limpia (ODS 7), ciudades y comunidades sostenibles (ODS 11) y acción por el clima (ODS 13), promoviendo soluciones innovadoras para garantizar un desarrollo urbano sostenible y mitigar el impacto del cambio climático.

El ODS 7, centrado en asegurar el acceso a una energía asequible, fiable, sostenible y moderna para todos, se vincula con las ciudades inteligentes mediante el impulso de tecnologías más limpias y eficientes. En el caso de los taxis y au-

tobuses autónomos, estos vehículos representan una oportunidad para reducir la dependencia de combustibles fósiles y promover la adopción de energías renovables en el transporte urbano. La validación de estos sistemas autónomos busca garantizar su eficiencia energética, reduciendo las emisiones de gases de efecto invernadero y contribuyendo así a los objetivos del ODS 7.

Por otro lado, el ODS 11 busca lograr que las ciudades y los asentamientos humanos sean inclusivos, seguros, resilientes y sostenibles. La integración de taxis y autobuses autónomos en el tejido urbano debe considerar aspectos de accesibilidad, seguridad y eficiencia en la movilidad. La reparación de planes en este contexto implica corregir posibles problemas de diseño, infraestructura o regulación que podrían afectar la implementación exitosa de estos sistemas de transporte autónomo. Asimismo, la adaptación de planes se hace necesaria para ajustar políticas urbanas y de transporte, fomentando la inclusión de estas tecnologías de manera equitativa para toda la comunidad.

En cuanto al ODS 13, que trata sobre la acción por el clima, los taxis y autobuses autónomos desempeñan un papel en la reducción de la huella de carbono y la mitigación del cambio climático. Al ser vehículos eléctricos o utilizar tecnologías más eficientes, contribuyen a la descarbonización del transporte urbano, apoyando la transición hacia una movilidad más sostenible. Sin embargo, es esencial validar su impacto ambiental real y reparar cualquier aspecto que pueda estar generando externalidades negativas, como la gestión adecuada de baterías o la optimización de rutas para reducir emisiones.

En conjunto, la validación, reparación y adaptación de planes en smart cities están intrínsecamente ligadas a estos ODS. La validación asegura la eficiencia energética, la reparación busca la sostenibilidad urbana y la adaptación se enfoca en mitigar el impacto ambiental, contribuyendo así a la construcción de entornos urbanos inteligentes, inclusivos y sostenibles.

Las ciudades inteligentes del futuro no solo deben ser tecnológicamente avanzadas, sino también social y ambientalmente responsables. Al abordar estos ODS en la planificación y ejecución de estrategias urbanas, se puede lograr un equilibrio entre la innovación tecnológica y el desarrollo sostenible, creando entornos urbanos que sean ejemplos de eficiencia, equidad y resiliencia para las generaciones venideras.