**Memoria de la Práctica 1 – Planificación Clásica con PDDL**

**1. Introducción**

**1.1 Objetivo de la Práctica**

Esta práctica tiene como objetivo aplicar distintos modelados de problemas de planificación automática utilizando PDDL. Se busca entender las capacidades, funcionalidades y limitaciones de los distintos tipos de modelos de planificación y su aplicabilidad en escenarios reales.

Este proyecto centrará los problemas en la gestión de un sistema de atención de emergencia de transporte de mercancía a través de drones. Posteriormente se probará el modelo con distintos planificadores para evaluar su rendimiento.

**Los objetivos específicos de la práctica incluyen:**

* Modelar el problema de planificación en PDDL clásico con restricciones STRIPS[[1]](#footnote-1).
* Crear un generador de problemas aleatorio donde se le pasará la cantidad de elementos que participarán en la creación.
* Evaluar el rendimiento de distintos planificadores automáticos.

**1.2 Estructura de la Documentación**

**Esta memoria está estructurada en las siguientes secciones:**

* **Introducción**: Presenta los objetivos de la práctica y una visión general de la documentación.
* **Desarrollo de la Práctica**: Explica el modelado del dominio en PDDL, la generación de problemas en Python y la evaluación de los planificadores.
* **Resultados y Análisis**: Muestra los experimentos realizados y una comparación del rendimiento de distintos planificadores.
* **Conclusiones**: Resume los hallazgos clave y propone mejoras futuras.
* **Anexos**: Incluye fragmentos de código, tablas de resultados y salidas de los planificadores.

**1.3 Contenido del Proyecto**

**El proyecto incluye los siguientes archivos y directorios:**

**PDDL**

* **parte1/pddl/dominio-drones.pddl – Modelo del dominio de planificación.**
* **parte1/pddl/problema.pddl – Instancia de problema generado como prueba**

**Generador de Problemas**

* **parte1/generador/generate\_problem.py – Script en Python para la generación de problemas.**
* **parte1/generador/test\_planiffier.py – Script en Python para generar gráficos de eficiencia de los planificadores dados.**

**Planificadores**

* **planificadores/ff – Planificador Fast Forward (FF).**
* **planificadores/lpg-td – Planificador LPG-TD basado en búsqueda local.**
* **planificadores/sgplan40 – Planificador SGPLAN40 basado en descomposición de problemas.**

**ProblemasGenerados**

* **almacenamiento de problemas generados por el generador de problemas**

**2. Desarrollo de la Práctica**

**2.1 Modelado del Problema en PDDL**

* **Descripción del dominio**:

**El dominio de esta primera parte es uno muy sencillo**.

En primer lugar, hemos definido una serie de tipos para identificar a los distintos tipos de actores que formarán parte del problema. Estos tipos son:

* dron: Es el instrumento de transporte de mercancías y el protagonista del desarrollo de este problema de planificación.
* brazo: Componente del dron con el que este puede interactuar para llevar las cosas de un lado a otro. Es independiente al dron ya que de esta forma, incluso aplicando funcionalidades equivalentes, es capaz de ser una aplicación mucho más escalable.
* persona: Usuario solicitante de suministros. Se localizan en algún sitio y necesitan un tipo de suministro capaz de ser transportado por los drones.
* localización: Posición o lugar física por la que comprendemos la capacidad de un dron de relacionarse con el ambiente.
* caja: Instrumento de transporte de suministros.
* contenido: Suministros

Tras especificar los tipos, hemos creado unos predicados para permitir al sistema interactuar entre sí. Estos son los predicados:

* **dron-en** ?d – dron ?l – localización: Indica si un dron está en una posición
* **caja-en** ?c – caja
* **persona-en** ?p - persona ?l - localización: Indica si una persona está en una posición
* **sostiene** ?d - dron ?b - brazo ?c - caja: Indica si un dron sostiene una caja con un brazo.
* **brazo-libre** ?d - dron ?b - brazo: Indica si un brazo de un dron está libre.
* **necesita** ?p - persona ?t - contenido: Indica si una persona necesita un cierto contenido.
* **tiene** ?p - persona ?t - contenido: Indica si una persona posee un cierto contenido.
* **contiene** ?c - caja ?t - contenido: Indica si una caja contiene un cierto contenido.

Finalmente, para poder realizar los cambios en el sistema, debe haber acciones que causen dichos cambios. Para ello, hacemos uso de las acciones, que son las siguientes:

* **Acción: coger**

**Descripción**

Un dron usa uno de sus brazos para recoger una caja en la misma localización.

**Precondiciones**

* El dron (?d) debe estar en la misma localización (?l) que la caja (?c).
* El brazo (?b) del dron debe estar libre.

**Efectos**

* El dron sostiene la caja con el brazo (sostiene ?d ?b ?c).
* La caja ya no está en la localización (not (caja-en ?c ?l)).
* El brazo utilizado ya no está libre (not (brazo-libre ?d ?b)).
* **Acción: volar**

**Descripción**

El dron se desplaza de una localización a otra.

**Precondiciones**

* El dron (?d) debe estar en la localización de origen (?from).

**Efectos**

* El dron deja de estar en la localización de origen (not (dron-en ?d ?from)).
* El dron pasa a estar en la nueva localización (dron-en ?d ?to).
* **Acción: entregar**

**Descripción**

Un dron entrega una caja a una persona que la necesita.

**Precondiciones**

* El dron (?d) debe sostener la caja (?c) con el brazo (?b).
* El dron (?d) y la persona (?p) deben estar en la misma localización (?l).
* La caja (?c) debe contener el contenido (?t).
* La persona (?p) debe necesitar el contenido (necesita ?p ?t).

**Efectos**

* La persona (?p) obtiene el contenido (tiene ?p ?t).
* El dron deja de sostener la caja (not (sostiene ?d ?b ?c)).
* La persona ya no necesita el contenido (not (necesita ?p ?t)).
* El brazo del dron queda libre (brazo-libre ?d ?b).
* **Ejemplos de instancias de problemas**:
  + Problema básico con pocas entidades.
  + Problema más complejo con múltiples localizaciones y restricciones.

**2.2 Generador de Problemas en Python**

* **Descripción del código**:

Para crear problemas, hemos usado el esqueleto del generador proporcionado y hemos gestionado algunos cambios para adaptarlo al dominio que hemos creados.

Este generador de problemas aleatorio tiene en cuanta una serie de parámetros dados por el usuario de la aplicación y genera un problema con la cantidad de actores y acciones solicitadas de forma completamente aleatoria.

Para generar correctamente el problema, hay una serie de funciones auxiliares que apoyan a su creación.

**LAS FUNCIONES AUXILIARES:**

**distance(location\_coords, location\_num1, location\_num2):** Función proporcionada por el esqueleto. Obtiene la distancia euclidiana entre dos ubicaciones. No la usaremos en esta parte del trabajo.

**flight\_cost(location\_coords, location\_num1, location\_num2):** Función proporcionada por el esqueleto. Caclula el coste de vuelo entre dos ubicaciones basado en la distancia. Llama a distance() para obtener la distancia entre ellos. Devuelve el valor de la acción basado en la distancia redondeada hacia arriba. No la usaremos en esta parte del trabajo.

**setup\_content\_types(options):** Función proporcionado por el esqueleto. Esta función auxiliar trata de generar de forma aleatoria el contendido de las cajas, siendo estos pasados como parámetro. Además, esta función se asegura de que al menos una caja tenga un elemento de los existentes en el sistema (en caso de haber comida y medicina, habrá una de cada en alguna de las cajas)

**setup\_location\_coords(options):** Función proporcionada por el esqueleto. Esta función asigna coordenadas aleatorias a las localizaciones (tipo hablado anteriormente). No la usaremos en esta parte del trabajo.

**setup\_person\_needs(options, crates\_with\_contents):** Genera necesidades aleatorias de contenido para las personas generadas en el problema. Esta función asegura de que no se le asignen más necesidades de las que debe cubrirse.

**FUNCIÓN PRINCIPAL/MAIN:**

La función principal permite, mediante la introducción de una serie de parámetros que indican los actores que formarán parte de un sistema, que este genere un problema aleatorio.

El programa es resistente a la no introducción de los parámetros.

* **Ejecución del generador**:
  + Ejemplo de uso con parámetros.
  + Tipos de problemas generados y análisis de su dificultad.

Para ejecutar correctamente el generador, es necesario llamarlo por la termina. Esto se hace mediante la llamada del archivo de Python y es necesario pasarle obligatoriamente los siguiente parámetros:

* --drones
* --carriers
* --locations
* --personas
* --crates
* --goals

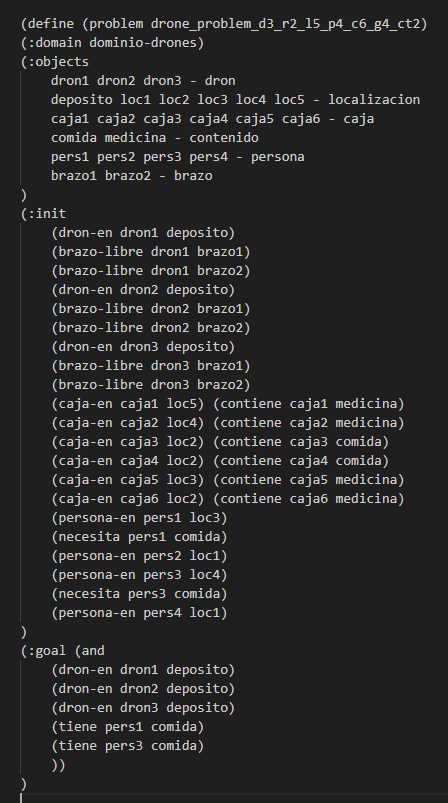
Y tras cada uno de esos parámetros, es necesario añadirle un número, que indicará la cantidad de ese tipo se crearán para el programa.

El problema se guarda en la carpeta problemasGenerados de la parte que le corresponda.

**EJEMPLO DE USO:**

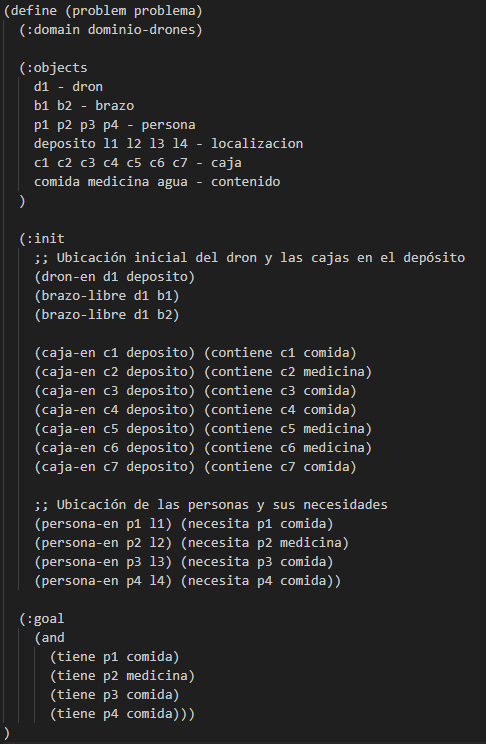
python parte1\generadores\generadorAleatorio.py --drones 3 --carriers 2 --locations 5 --persons 4 --crates 6 --goals 4

Mediante el anterior comando, hemos llamado al programa desde la carpeta general del proyecto y genera un archivo tal que:



* 1. **Ejemplo de problema:**

**El problema que vamos a tratar es el siguiente:**

****

**Para entender el problema, vamos a desglosarlo y comentarlo:**

En dicho problema, podemos identificar estos objetos:

* 1 dron
* 2 brazos (pertenecen al dron)
* 4 personas
* 5 localizaciones (de la cual una es el depósito)
* 7 cajas
* 3 tipos de contenido

Inicializamos los datos tal que:

* El dron empieza en el depósito y tiene dos brazos.
* Todas las cajas están en el depósito y contienen: Comida, medicina, comida, comida, medicina, medicina, comida (respectivamente a las 7 cajas).
* Las personas aparecen en las localizaciones, que dado nombres como:

(persona1, persona 2, …, persona4 y localización1, localización2, …, localización4) están todos en las localizaciones coincidentes con su número y necesitan comida, medicina, comida, comida también respectivamente al número de la persona.

El objetivo del problema sería conseguir los suministros que piden las personas. Estos suministros son:

* persona1: comida
* persona2: medicina
* persona3: comida
* persona4: comida
  1. **Evaluación de Planificadores**

Para evaluar los planificadores, hemos usado un tester de planificadores que realizaba soluciones a problemas generados aleatoriamente de dificultad creciente hasta que la búsqueda de una solución superase los 60 segundos.

El tester en cuestión, era un programa de Python y funcionaba de la siguiente manera:

**FUNCIONES AUXILIARES:**

**time\_limit(seconds):** Iniciaba un contador de duración “seconds” que detenía la ejecución de la búsqueda de soluciones una vez superaba dicha cantidad.

**find\_newest\_problem\_file(directory=”parte1/generadores”):** No usada. Buscaba el último archive cuyo nombre fuese (“drone\_problem\_\*.pddl”), siendo el asterisco un sustituto donde podrían entrar cualquier combinación de caracteres.

**generate\_problem(drones, carriers, locations, presons, crates, goals):** Llama al archive de generación de problemas que se ha mencionado anteriormente y le pasa exactamente los mismos parámetros que se le están pasando para que genere un problema. Una vez está el problema creado, devuelve la ruta. En caso de haber un error, no devuelve nada.

**run\_planner(domain\_file, problem\_file,planner\_path, time\_limit\_seconds=60):** Dado un dominio, un problema, un planificador y una cantidad determinada, hace que el planificador genere una solución del problema en base al dominio con un límite de 60 segundos. En caso de que no lo tenga, este devuelve la solución y el tiempo que le ha costado, y en caso de que no, devuelve un None, especificando que no se ha encontrado una solución, y el tiempo dado, como para señalizar que se ha excedido.

**delete\_problem\_file(problema\_fila):** Borra todos los archivos cuyo nombre sea “dron\_problem\_\*.pddl”, donde el asterisco es un sustituto donde podrían entrar cualquier combinación de caracteres.

**plot\_results(sizes, times, solutions\_found, max\_size, planner\_name):** Hace gráficos respecto a las soluciones encontradas de un problema. Este gráfico será posteriormente guardado como un png.

**FUNCIÓN PRINCIPAL/MAIN:**

Es una función que aplica problemas de coste incremental hasta que excede alguno de los límites establecidos.

Los parámetros intraducibles son:

* --planner: Se le introduce la ruta en la que está alojado un planificador para ponerlo a prueba
* --domain: Se le pasa la ruta del dominio del problema para que sepa el planificador que normas, predicados y tipos tiene el problema
* --start-size: Seguido de un número establece el tamaño inicial del problema.
* --max-size: Seguido de un número establece el tamaño máximo del problema. Uno de los límites que puede llegar a ser superado y limitar la continuación del programa.
* --timeout: Seguido de un número indica el tiempo máximo que puede tomar un planificador para solucionar un problema.
* --continue-on-fail: Continúa las pruebas aunque haya fallos consecutivos.

**Los planificaciones que hemos usado son**:

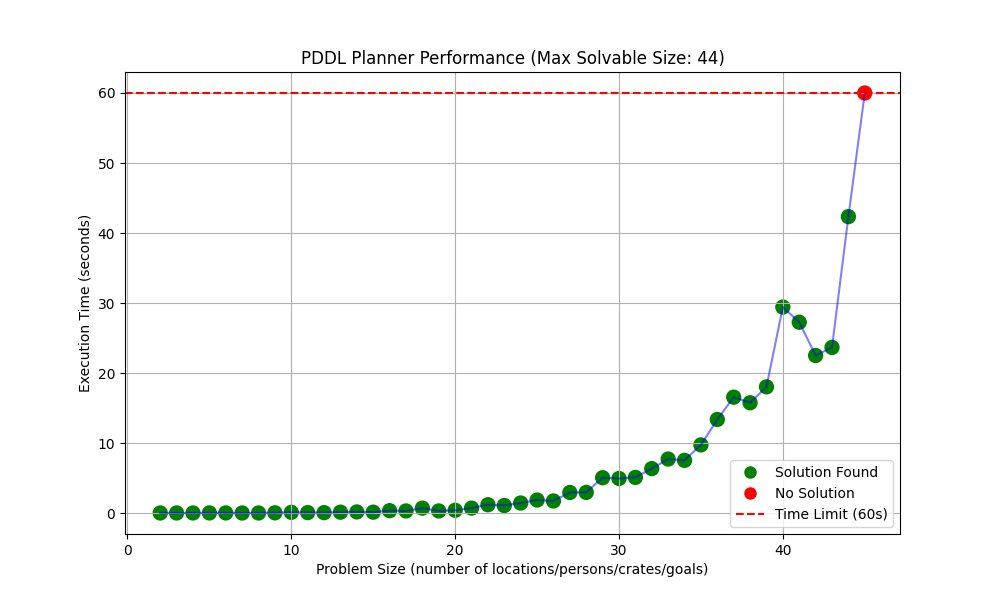
# FF:

Cualidades:

* Generador de planes rápidamente sin optimizar en costos.
* Basado en heurísticas de relajación, donde prioriza buscar soluciones con pocas acciones, a acciones de bajo costo.
* Compatible con PDDL STRIPS
* No maneja costes ni duraciones. Por lo que es muy rápido para llegar a soluciones, pero están poco optimizadas

Limitaciones:

* Al no considerar costes ni duraciones de acciones, genera soluciones que pueden llegar a ser caros.
* Debido a su forma de aplicar la heurística, si no encuentra mejoras inmediatas, puede quedarse atascado.
* No soporta acciones durativas ni concurrencias, lo que lo hace ineficiente en dominios



Cómo podemos observar en la búsqueda de soluciones de problemas dados, encuentra el límite en problemas se tamaño 44.

Solución:

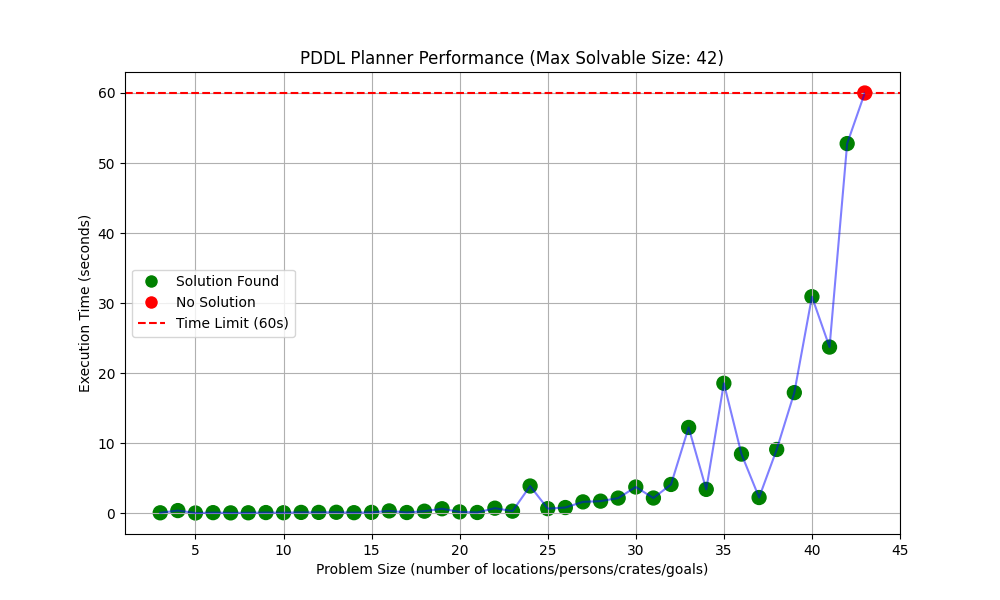
# LPG-TD:

Cualidades:

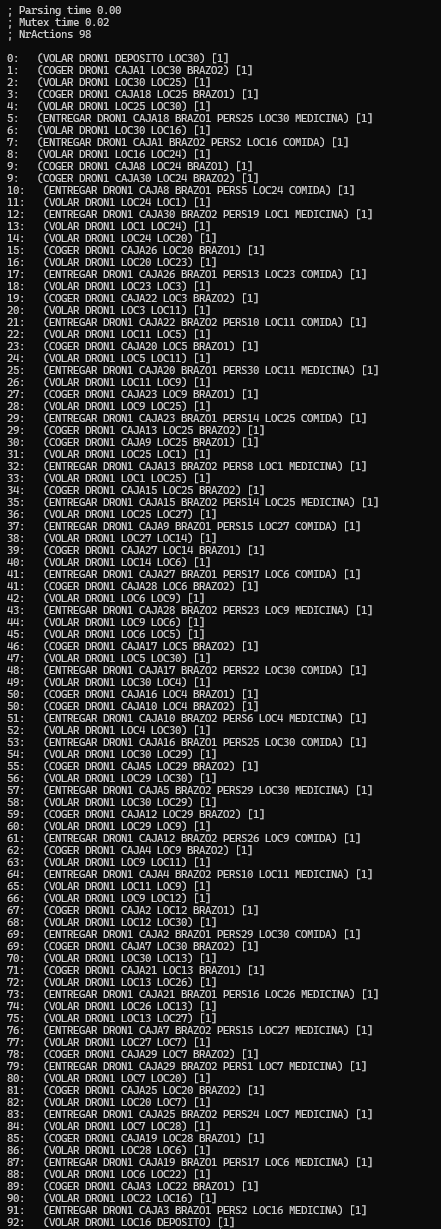
* Planificador basado en búsqueda local con optimizaciones heurísticas
* Soporta costes y duraciones de acciones
* Genera múltiples planes en paralelo y permite optimización en distintos criterios
* Compatible con PDDL STRIPS
* Es supuestamente eficiente en problemas grandes y con restricciones de tiempo

Limitaciones:

* Puede ser algo lento en problemas sin restricciones temporales
* Basado en su búsqueda local, puede generar soluciones subóptimas
* Puede requerir ajustes manuales para mejorar soluciones en ciertos dominios



El gráfico nos muestra que es capaz de encontrar resultados hasta alcanzar un tamaño de 42.

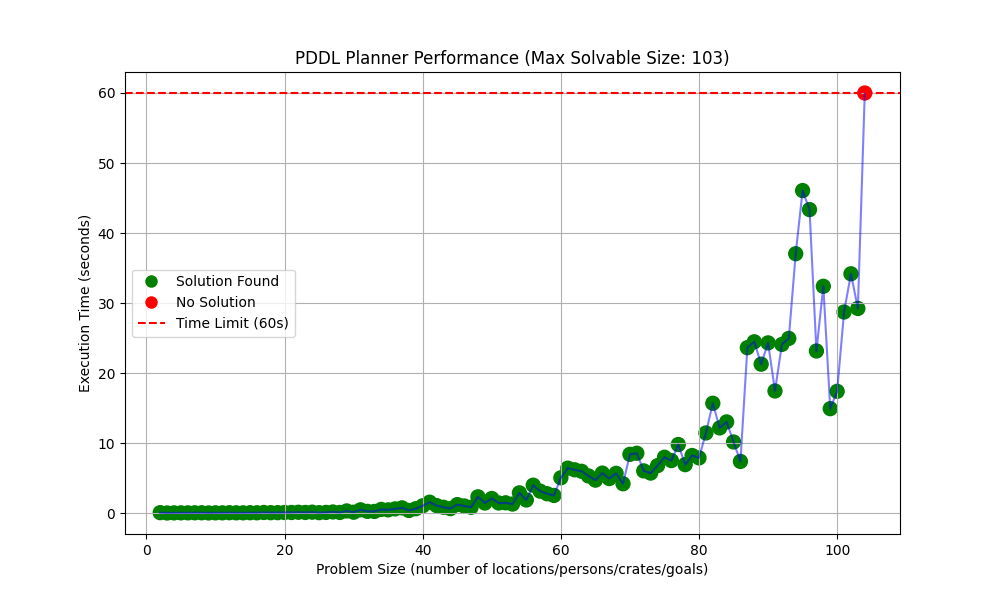
Solución:

# SGPLAN40:

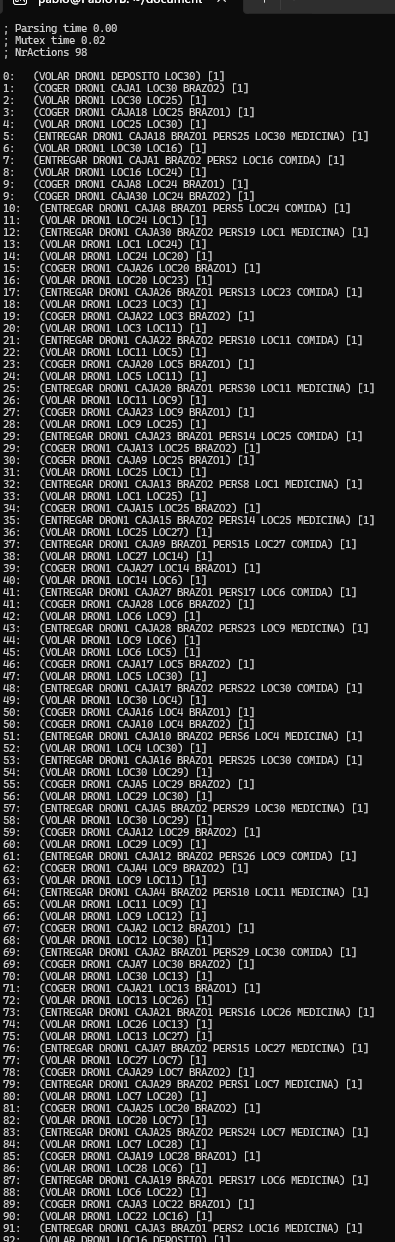
Cualidades:

* Planificador basado descomposición de problemas en subproblemas más pequeños
* Soporta acciones durativas, concurrencia y restricciones temporales
* Compatible con PDDL STRIPS
* Eficiente en dominios con restricciones y planificación multi-objetivo.
* Maneja bien problemas de gran escala donde otros planificadores se pueden quedar atascados

Limitaciones:

* Su descomposición de problemas puede generar planes inconsistentes si los subprogramas no están bien coordinados
* No siempre encuentra la solución globalmente más óptima debido a su método de segmentación
* Puede ser sensible a la estructura del problema

Superando las capacidades de los dos planificadores anteriores, este modelo es capaz de solucionar problemas hasta de un tamaño de 104.

Solución:

**Para poder comparar correctamente los distintos modelos, he seleccionado un problema de tamaño 30 para que lo resuelvan todos y lo he dejado en el apartado final de cada planificador con “solución”.**

* **Comparación de rendimiento**:

La lectura del rendimiento de todos los planificadores, apuntan a que el mejor para generar una planificación en este contexto, es el SGPLAN-40, que es capaz de llegar a encontrar soluciones inferiores 60 segundos de tamaño de hasta 103.

Además, podemos descartar de forma concluyente al planificador “ff”, ya que no solo es por poco igual de lento que el LPG-TD, sino que tiene muchas menos funcionalidades y genera más pasos de los que generan los otros dos planificadores.

Por otra parte, el LPG-TD, tiene la capacidad de buscar soluciones de forma simultánea, lo que puede impulsar su uso y de dichas soluciones, comparar para ver la óptima.

Sin embargo, y volviendo al planificador inicial, es el más rápido, también optimiza la solución con el fin de encontrar una, aunque no siempre óptima, optimizada.

**4. Conclusiones**

1. STRIPS: Restricciones en problemas de planificación. Esto se aplica como: Tiene precondiciones, que de ben cumplirse antes de ejecutar una acción; efectos, que son los cambios que aplican las funciones; sin condiciones negativas, no se puede definir una condición como opuesta a una existente; sin efectos condicionales, no se puede definir bifurcaciones en la toma de decisiones de una acción concreta; y no tiene cuantificadores ni expresiones complejas, expresiones como ∀ (para todo) o ∃ (existe). [↑](#footnote-ref-1)