GENERACIÓN DE POLÍTICAS PARA PLANIFICACIÓN HEURÍSTICA MEDIANTE METACLASIFICADORES

PROYECTO FIN DE CARRERA

Alberto Garbajosa Poderoso

Ingeniería en Informática
Escuela Politécnica Superior
Universidad Carlos III de Madrid

12 de junio de 2013

GENERACIÓN DE POLÍTICAS PARA PLANIFICACIÓN HEURÍSTICA MEDIANTE METACLASIFICADORES

Proyecto fin de carrera

Grupo PLG (Planning & Learning Group)
Departamento de Informática

Director
Tomás Eduardo de la Rosa Turbides
Tutor
Raquel Fuentetaja Pizán

Ingeniería en Informática
Escuela Politécnica Superior
Universidad Carlos III de Madrid

12 de junio de 2013

Título: Generación de políticas para planificación heurística mediante metaclasificadores.

Autor: Alberto Garbajosa Poderoso.

Tutor: Raquel Fuentetaja Pizán.

Director: Tomás Eduardo de la Rosa Turbides.

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día ___ de _____ de 20___ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Dedicado a mi familia.

Resumen

La planificación automática es una de las ramas de la inteligencia artificial, que tiene por finalidad la generación de planes de manera automática. Para ello, se usan a menudo funciones heurísticas que guían el proceso de decisión de qué operador escoger para transitar de un estado a otro, con la finalidad de alcanzar las metas propuestas con el menor coste posible.

Estudios anteriores han demostrado que la inclusión del aprendizaje automático como soporte a la decisión del planificador, genera buenos resultados. Así, mediante ejemplos de aprendizaje se conseguía generar un árbol de decisión, que era suministrado al planificador.

Este proyecto de investigación tiene como finalidad principal evaluar si esta tendencia positiva se mantiene si, en vez de usar un único árbol, se emplean combinaciones de ellos al mismo tiempo. Mediante técnicas de bagging, se consigue dividir el conjunto de entrenamiento inicial, para crear varios árboles de decisión que recibe el planificador. Éste, a su vez, emplea distintos algoritmos de combinación de la información aportada por los árboles.

El estudio experimental ha demostrado varios aspectos relevantes. En primer lugar, el uso de estas técnicas ha supuesto una mejora respecto al uso de un sólo árbol, resolviendo en ocasiones más problemas con menor coste. Pero esto no es así en todos los dominios evaluados, existiendo casos para los cuales no existen diferencias significativas en los planes generados. En segundo lugar, se ha de considerar el hecho de que el uso de varios árboles incrementa los tiempos tanto de entrenamiento como de validación, siendo más notables en el primer caso. Lo mismo ocurre con la memoria (principal y secundaria) requerida para el sistema, que se verá aumentada cuantos más árboles se quieran emplear de forma simultánea.

Palabras clave: planificación heurística, aprendizaje automático, bagging, árbol de decisión, ROLLER.

Abstract

Automated planning is an artificial intelligence branch, whose objective is generating plans automatically. Heuristic functions are often used to help choosing which operator is selected to transit from an state to another, in order to reach the proposed goals with the lower cost.

Previous studies demonstrated that the use of machine learning combined with planners generates good results. An example is the use of decission trees to guide the planner.

The objective of this project is to evaluate if this positive trend is maintained using combinations of several decission trees at the same time, instead of using one single tree. Through the use of bagging techniques, the initial training set is split in order to create several decision trees which are received by the planner. This planner uses different algorithms which combine the information provided by the trees.

The experimental study has demonstrated some relevant aspects. First, the use of these techniques has involved an improvement over the use of a single tree, solving more problems with lower cost for some domains. Second, the use of several trees increases the training and testing time. The same thing happens with the memory (main and secondary) required by the system, which will be increased as more trees are used at the same time.

Keywords: heuristic planning, machine learning, bagging, decision tree, ROLLER.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Resumen: En este capítulo se hace una introducción de la materia del proyecto, así como los objetivos que se buscaban con su realización. También se comenta brevemente la estructura que seguirá el resto de este documento.

1.1. Introducción

La planificación automática es una de las áreas de la inteligencia artificial, cuyo objetivo principal es crear programas capaces de generar planes de actuación para cualquier dominio con unas acciones predefinidas, en el que partiendo de un estado inicial se deben alcanzar las metas propuestas, empleando dichas acciones.

La dificultad radica en que el tipo de problemas que se buscan resolver mediante planificación automática son lo suficientemente complejos como para afirmar que la generación de planes es una tarea muy cara computacionalmente.

Por ello, la investigación en este campo se centra no sólo en ser capaces de obtener planes, sino en que estos planes sean buenos, de manera que el esfuerzo empleado en su generación se vea recompensado por la utilidad que presenten. La bondad de dichos planes se puede medir de acuerdo a ciertos criterios, que al final se podrían resumir en dos valores, como en cualquier otro proyecto de ingeniería: coste y tiempo.

El coste puede hacer alusión a la capacidad de cómputo de las máquinas empleadas para generar los planes o a la cantidad de memoria requerida, elementos que pueden ser clave a la hora de llevar a cabo un proyecto de forma satisfactoria. Sin embargo, en este proyecto la estimación del coste va más encaminada a tener una medida de lo bueno o malo que es un plan: el número de pasos necesarios para llegar a las metas, partiendo del estado inicial.

Esto no es algo trivial; puesto que, aunque para problemas muy sencillos se pueda conocer el plan óptimo (el que llega a las metas en el menor número de pasos posible), los problemas para los que se hace patente la importancia de la planificación automática son aquéllos para los que no se conoce una solución óptima. Es entonces cuando cobra sentido el hecho de decir que un planificador genera un plan mejor que otro porque haya sido capaz de dar una solución válida con menor número de acciones.

Hay que mencionar también que, para algunos casos de planificación, se formalizan los dominios de manera que determinadas acciones tienen un coste asociado (de ejecución) mayor que el de otras. Esto es, que partiendo de un estado inicial Q, ejecutar la acción A2 cueste más que ejecutar la acción A1, independientemente de lo cerca o lejos que el estado final esté de las metas. Es entonces cuando por encima del número de acciones, habrá que priorizar la elección de éstas, de forma que no sólo permitan llegar a las metas, sino que además el coste adicional necesario para ejecutarlas sea el menor posible.

Por otro lado, una medida del tiempo necesario para generar estos planes puede ayudar a tener una mejor visión de cómo de eficiente es el planificador creado.

A la hora de crear nuevos planificadores, se buscan nuevas ideas con las cuales se espera que se puedan llegar a obtener mejoras significativas en la calidad de los planes obtenidos y en el tiempo en el que éstos son generados. Las decisiones que se tomen irán encaminadas a buscar un equilibrio entre ambos valores o, cuando esto no sea posible, fomentar aquél que pueda ser más conveniente para cada caso.

Partiendo de esta idea, se han implementado multitud de planificadores con diferentes formas de abordar el tema. Actualmente se emplean sistemas de planificación heurística, que utilizan unas funciones (heurísticas) encargadas de orientar al planificador a la hora de explorar los distintos caminos posibles en la búsqueda de soluciones. Igualmente, las funciones heurísticas implementadas por los planificadores son muy diversas, y de su calidad dependerá en gran medida el éxito de esos planificadores.

El aprendizaje automático puede ser de utilidad para la planificación heurística; de manera que, partiendo de un conjunto de entrenamiento formado por planes anteriores, se consiga generar un clasificador que servirá de apoyo al planificador en su labor de búsqueda de soluciones.

En base al párrafo anterior, con este proyecto se ha profundizado en el planificador **ROLLER**, publicado en 2008 por Tomás de la Rosa y Sergio Jiménez, miembros del grupo PLG (Planning & Learning research Group) de la UC3M (Universidad Carlos III de Madrid). La particularidad de este planificador es que emplea un árbol de decisión relacional como complemento a la tarea de planificación heurística, lo que demostró generar buenos resultados respecto a otros planificadores. Así, ahora se ha querido comprobar

cómo respondería este planificador si en vez de tener un sólo árbol se tuvieran varios a la vez, para lo cual se han empleado técnicas de bagging, así como diferentes políticas de combinación de la información aportada por cada uno de los clasificadores construidos.

Durante el resto del documento, se explicarán las bases teóricas en las que se fundamenta este proyecto, se detallarán los métodos empleados tanto para generar los árboles de decisión, como para combinarlos de manera que puedan servir de soporte para el planificador. Se hará especial hincapié en el capítulo de experimentación, en el cual se evaluará el comportamiento del sistema para varios dominios con diferentes políticas para la creación y el uso de los árboles, así como la mejora del rendimiento de estos resultados respecto a los obtenidos por ROLLER "clásico" (empleando un solo árbol). Finalmente, se mostraran las conclusiones más relevantes del proyecto y se ofrecerán algunas vías de investigación para futuros proyectos.

1.2. Objetivos

El objetivo principal que se ha buscado con la realización de este proyecto es el de evaluar si los resultados positivos de ROLLER podían ser mejorados utilizando conjuntos de clasificadores. Es decir, si con un árbol de decisión, los planes obtenidos mejoraban significativamente respecto a los generados por otros planificadores, ¿se mantendrá esta tendencia si, en vez de un solo árbol, se emplean combinaciones de varios de ellos? De esta manera, el objetivo principal es ampliar el planificador existente, permitiendo la inclusión de nuevos algoritmos que sean capaces de desarrollar diferentes políticas de generación y combinación de árboles de decisión.

La solución deberá permitir, por tanto, crear un número variable de árboles de decisión para un dominio dado y varias opciones de combinación entre ellos. El número de árboles estará limitado por la potencia de la máquina que se utilice, ya que será necesario cargarlos en memoria al mismo tiempo.

Una vez desarrollada la ampliación, el siguiente objetivo será el de verificar si se ha mantenido la tendencia positiva anterior. Así, se querrá comprobar si existe una relación entre la bondad de los planes generados y el uso de más o menos árboles de decisión combinados.

De forma resumida, las ideas fundamentales que se buscan resolver son:

1. Plantear la manera de dividir el conjunto de entrenamiento inicial.
2. Implementar métodos que permitan realizar la división planteada, basados en bagging.
3. Ser capaces de generar varios árboles de decisión en una misma ejecución del sistema, empleando los distintos conjuntos de entrenamiento creados.

4. Diseñar varias políticas de combinación de la información aportada por los árboles.
5. Realizar experimentos para varios dominios, bajo condiciones de prueba diferentes.
6. Evaluar la mejora de este sistema respecto a ROLLER con un solo árbol.
7. Comparar los resultados entre experimentos con varios árboles.

En cuanto a objetivos de diseño, la realización de este proyecto está limitada por el planificador de partida (ROLLER) con el que tiene que ser compatible. No se trata, por tanto, de una herramienta nueva e independiente, sino más bien de una ampliación de las características de la versión previa. Por ello, la mayor parte del código desarrollado ha de tener en cuenta el funcionamiento de ROLLER: el tratamiento de datos que hace, la nomenclatura de los algoritmos, las entradas y salidas de las funciones, etcétera. El lenguaje de programación empleado será C.

Por otro lado, el proceso de experimentación se ha decidido automatizar, de manera que se hagan las llamadas pertinentes al planificador para cada dominio y problema correspondiente. Esta parte no necesita cumplir el objetivo anterior, puesto que se utiliza externamente de manera independiente. En este caso, se empleará el lenguaje de programación Python.

Finalmente, respecto a otros posibles objetivos a tener en cuenta, no se necesitará ser demasiado estricto, dada la naturaleza del proyecto realizado. Por ejemplo, la rapidez del planificador, aunque es una medida cuantificada en los experimentos, no es un factor clave para el desarrollo; ya que, al usar más árboles que ROLLER, es normal que tarde más, tanto en crearlos como en usarlos. Por otro lado, el tiempo que tarde en resolver un problema, dependerá no sólo del planificador, sino del problema en sí, del dominio, del conjunto de entrenamiento utilizado, etcétera. Pese a ello, como se comentará más adelante, en la fase de experimentación se limitará el tiempo por problema, de manera que se descarten aquéllos que sobrepasen el máximo permitido.

Además, dado que es un tema de investigación, las conclusiones del proyecto irán enfocadas más a los resultados que a la aplicación en sí. De esto se deriva que tampoco se busquen como objetivos principales la facilidad de uso del sistema o la flexibilidad del código implementado, a pesar de que en algunos momentos se puedan tener en cuenta.

1.3. Estructura de la memoria

En este apartado se va a comentar la estructura que se ha seguido durante el resto del documento, de manera que la información que contiene pueda

ser encontrada con facilidad.

En primer lugar, se comenzará por explicar lo que se ha denominado el "Estado de la cuestión". En ese capítulo se hará referencia a varias materias, directamente relacionadas con este proyecto, cuyo entendimiento se hace esencial para comprender mejor lo que se ha intentado demostrar con su realización.

Así, se definirán las bases de la planificación automática y del aprendizaje automático, como ramas del campo de la inteligencia artificial. Se explicará brevemente el funcionamiento de varios planificadores. También se hará mención a otro de los programas auxiliares (TILDE) empleados en el proyecto. Finalmente, se explicará la aproximación de partida del proyecto (ROLLER).

Posteriormente, se detallará el desarrollo del proyecto, indicando la arquitectura general del sistema y explicando el funcionamiento del código diseñado.

Más adelante, se profundizará en los experimentos realizados indicando los dominios evaluados, los problemas de partida, las condiciones de las pruebas (como pueda ser, por ejemplo, el tiempo máximo dedicado a cada problema) y se mostrarán gráficos comparativos para las distintas aproximaciones y dominios.

Después de mostrar los resultados obtenidos, se explicará cuáles han sido las conclusiones más relevantes que se han podido extraer de los experimentos realizados.

Finalmente, se comentarán algunas ideas de mejora para continuar en el futuro o para abordar la cuestión desde otros puntos de vista alternativos.

Capítulo 2

Estado de la cuestión

Resumen: En este capítulo se comentan las bases teóricas en las que se fundamenta este proyecto, de manera que puedan ser de utilidad como punto de partida para iniciarse en la materia y entender la motivación del mismo. Así, partiendo de lo más general y llegando hasta los detalles más propios del proyecto, se detallarán algunos aspectos relevantes de la planificación automática, del aprendizaje automático, de algunos planificadores y del funcionamiento de los programas empleados para el proyecto (ROLLER y ACE).

2.1. Planificación automática

La Planificación Automática es una de las ramas de la Inteligencia Artificial, que se centra en la generación de planes de manera automática, los cuales se compondrán de una serie de acciones que han de ejecutarse en un orden determinado. De forma general, la finalidad de estos planes es que puedan ser utilizados posteriormente por dispositivos automatizados, como pueden ser robots o agentes inteligentes.

Para generar los planes, se emplean programas informáticos denominados planificadores, que se encargan de decidir qué acciones hay que realizar y en qué orden, para conseguir alcanzar el objetivo deseado. De forma más precisa, un planificador recibe tres entradas formalizadas (normalmente, empleando lógica de predicados), que son:

1. **Estado inicial** en el que se encuentra el "mundo" para un dominio determinado.
2. Conjunto de **metas** que se quieren alcanzar.
3. Conjunto de **acciones** (operadores) posibles que pueden ser realizadas, y que permiten pasar de un estado a otro, para intentar alcanzar las metas.

El estado inicial junto con las metas, son propios de cada problema que se quiera resolver; mientras que, el conjunto de acciones, está determinado por el dominio.

Las acciones generalmente tienen un conjunto de precondiciones necesarias para que pueda ejecutarse dicha acción y un conjunto de postcondiciones (añadidos y borrados) que definen el estado del sistema tras realizarla. De esta manera, para un estado, no podrán realizarse todas las acciones, sino sólo aquéllas para las que se cumplan las precondiciones.

La tarea del planificador consiste en ir decidiendo, de entre todas las acciones posibles en un momento determinado, cuál de ellas realizar: el plan estará formado por la secuencia de acciones escogidas.

Es fácil darse cuenta de que, para dominios muy complejos, en los que el número de acciones que se pueden realizar en un momento determinado es grande, el hecho de decidir qué acción es la que debe ser escogida para llegar antes a la meta se vuelve una tarea complicada, puesto que muchas veces el hecho de escoger una u otra acción no es evidente, ya que ésta no aporta por sí misma nada de información sobre si se llegará antes a la meta o no. Si a ello le sumamos la tarea de resolver problemas complicados, en los cuales las metas se encuentren muy alejadas del estado inicial (y, por tanto, los planes para resolverlo serán largos), las posibles combinaciones de acciones crecen enormemente, haciendo que la tarea del planificador se complique.

Una manera de hacer frente a las decisiones de qué acción escoger para cada estado, es usar lo que se denomina **Planificación Heurística**. Se basa en emplear funciones (heurísticas) que ofrecen una medida de "lo lejos" que estaríamos de la meta desde un estado determinado. De esta manera, si se decide seguir la indicación de la heurística, se optará por escoger aquellas acciones con las que más nos acerquemos a la meta.

Las funciones heurísticas, sin embargo, no indican verdades absolutas, ya que a la hora de resolver un problema de planificación no se conoce su solución, puesto que es precisamente lo que se está intentando obtener. La planificación heurística se basa en resolver problemas "relajados" (típicamente se obtienen suprimiendo el conjunto de eliminados de las postcondiciones) para los cuales es fácil encontrar alguna solución. A la hora de decidir qué acción escoger para un problema determinado, se calcula para cada acción disponible el coste a la meta para el problema relajado, escogiendo la que minimice el coste. Por ello, cuanto más parecido exista entre el problema original y el problema relajado, mayor será la fiabilidad ofrecida por la heurística empleada.

Recapitulando, hasta ahora se ha visto como la finalidad de la planificación automática es crear listas ordenadas de operadores (planes) que resuelvan un problema y cómo las funciones heurísticas pueden ayudar al planificador a explorar el árbol de búsqueda, ofreciendo una orientación de qué nodo sería más conveniente expandir (qué operador/es escoger para cada

estado). Pero hasta ahora no sé ha hablado de cómo es ese árbol de búsqueda.

De forma simplificada, se puede entender un árbol de búsqueda como una estructura de datos, en la que el nodo raíz se corresponde con el estado de partida. Para cada operador que pueda ser seleccionado desde dicho estado, se generará un nodo hijo. A su vez, cada uno de ellos tendrá tantos hijos como operadores puedan ser escogidos desde ese estado, etcétera.

Puede entenderse la planificación automática como el hecho de optimizar la búsqueda dentro del árbol, permitiendo establecer una secuencia de operadores entre el estado inicial y los nodos (estados) meta. Es decir, se podrá llegar a las metas a través de diferentes caminos, siendo la tarea de un buen planificador escoger los más cortos (o de menor coste) en la medida de lo posible.

Teniendo esto en cuenta, se pueden distinguir de forma general dos métodos de búsqueda, siendo el primero de ellos el más utilizado por los planificadores heurísticos:

1. **Búsqueda hacia delante**: el nodo raíz del árbol de búsqueda se corresponde con el estado inicial del problema que se quiere resolver. La búsqueda consiste en expandir los nodos que nos acerquen al cumplimiento de las metas, hasta dar con la solución. Los nodos hijo serán aquéllos que se han generado tras aplicar unos operadores desde el padre, cumpliendo éste último las precondiciones de dichos operadores.

2. **Búsqueda hacia atrás**: el nodo raíz del árbol de búsqueda se corresponde con el estado meta del problema que se quiere resolver. Se expanden nodos hasta que se dan por satisfechas las condiciones del estado inicial del problema, momento en el que se da por terminada la búsqueda. Para expandir un nodo empleando un operador, a partir de un estado, este último debe cumplir las postcondiciones del operador.

2.2. Planificadores

En esta sección se van a comentar las características principales de dos planificadores: FF y LAMA. Existen más, pero en este documento se explican sólo estos dos por estar relacionados con el proyecto desarrollado (las pruebas que se realizaron con ROLLER mostraron una comparativa con resultados generados por estos planificadores).

2.2.1. FF

FF (Hoffmann y Nebel, 2001) (*Fast-Forward*, "avance rápido" en español) es un planificador independiente del dominio programado en C, cuyo funcionamiento consiste en buscar hacia delante en el espacio de estados. Para ello, emplea una función heurística que guía este proceso, obteniendo

una estimación de la distancia que faltaría a las metas si se tomara uno u otro camino dentro del árbol de búsqueda.

La función heurística estima las distancias en base a problemas relajados, en los cuales se han desestimando las listas de "borrados" en las postcondiciones. De esta forma, la solución al problema relajado será lo que se denomina "plan relajado" y la función heurística devolverá el número de pasos mostrados por dicho plan, lo que ayudará a estimar lo cerca que se está de las metas.

Como algoritmo de búsqueda, FF utiliza una versión de búsqueda en escalada denominada EHC (*Enforced Hill Climbing*), que realiza una búsqueda local en amplitud hasta encontrar un nodo intermedio que mejore el valor heurístico.

2.2.2. LAMA

LAMA (Richter y Westphal, 2010) es un planificador que emplea búsqueda heurística hacia delante. Su característica principal es que usa fórmulas proposicionales, denominadas "landmarks", que deben ser ciertas para cada solución de la tarea de planificación. Emplea variables no binarias y varias heurísticas sensibles al coste (tienen en cuenta los costes de ejecutar un operador u otro), de manera que se centra en tratar de encontrar aquellos planes que tengan asociado un menor coste total.

El algoritmo de búsqueda es $WA*$ ($A*$ con función de evaluación $W\Delta h(n)+g(n)$). La búsqueda $A*$ con peso (w) se usa con valores de peso decrecientes de forma iterativa, de manera que el planificador continua buscando mejores planes (con costes más bajos) hasta que la búsqueda concluye.

LAMA combina estas heurísticas sensibles al coste con la heurística usada por el planificador FF.

2.3. Aprendizaje automático

El Aprendizaje Automático es una de las ramas de la Inteligencia Artificial, que tiene por finalidad crear programas informáticos con la intención de conseguir que una máquina (o un agente, robot, etc.) sea capaz de "aprender" por sí misma la mejor manera de solucionar los problemas que deba afrontar.

Se emplea generalmente para automatizar la resolución de problemas cuya solución sería costosa de obtener manualmente. Por ejemplo diagnósticos médicos, análisis bursátiles, clasificación de secuencias de ADN, motores de búsqueda, comportamientos de robots, detecciones sísmicas, reconocimiento de imágenes por satélite: con fines militares o civiles (agricultura, geografía, planificación urbanística...), reconocimiento de caras, de huellas, de voz, etcétera.

Dentro del aprendizaje automático, el aprendizaje inductivo se basa en

entrenar a la máquina con multitud de ejemplos relacionados con la tarea que debe ser capaz de resolver, de manera que sea capaz de generalizar cierta información que de ellos puede ser extraída. En base a esto, la máquina creará sus propias reglas de conocimiento que le permitan posteriormente afrontar nuevos problemas con rapidez y eficacia.

La capacidad de aprendizaje, por tanto, estará limitada por el número y variedad de los problemas con los que se realice el entrenamiento, que le permitan inferir diferentes reglas.

Los problemas deberán estar formalizados, de modo que el programa informático sea capaz de comprenderlos. Éste, recibe como entrada cada problema, que consistirá en una serie de valores. Para ese proceso de formalización, se hace necesario comprender el dominio en el que se quiere aplicar el aprendizaje automático.

Entrando más en detalle, se pueden distinguir varios tipos de aprendizaje, que se realizan de forma diferente. El aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje por refuerzo son algunos de ellos. En la siguiente subsección se explicará el primero de los mencionados, por estar directamente relacionado con el proyecto.

2.3.1. Aprendizaje supervisado

El aprendizaje supervisado se llama así porque se conoce la salida correspondiente a la combinación de entradas para cada problema de entrenamiento utilizado. Debido a esto, es frecuente la necesidad de contar con "expertos" en la materia que se vaya a tratar, que serán los que proporcionen los datos de partida o al menos validen que los datos que se están empleando son fiables, antes de comenzar el aprendizaje.

Esta "salida" correspondiente a una combinación de entradas se suele denominar "clase", y es la que distingue a un tipo de elementos, que tienen algo en común, respecto a otros elementos, que pertenecerán a una clase diferente. La clase puede ser un valor numérico o una etiqueta.

El objetivo del aprendizaje, por tanto, será conseguir que el sistema adquiera las habilidades pertinentes para poder distinguir de manera adecuada cuándo unos ejemplos pertenecen a una clase o a otra.

El aprendizaje supervisado se emplea principalmente para dos tipos de tareas: predicción y **clasificación**.

Con la predicción se parte de un conjunto de datos, a partir de los cuáles el sistema tiene que establecer relaciones de manera que sea capaz de obtener reglas que definan su disposición. Para ello, se emplean comúnmente técnicas de regresión (lineal o no lineal, árboles de predicción, etcétera). Así, una vez que se ha entrenado el sistema, se tratará de determinar (*predecir*) el valor de una o más variables para un ejemplo dado, teniendo en cuenta las reglas aprendidas a partir del conjunto de datos con el que ha sido entrenado.

Mediante la clasificación, por otro lado, se trata de dividir los datos en varios conjuntos mutuamente excluyentes, donde los miembros de un mismo grupo estarán próximos entre ellos, y alejados del resto de grupos. La tarea del aprendizaje consiste en escoger los grupos, para posteriormente ser capaz de clasificar un ejemplo dentro de uno u otro grupo.

Algunos ejemplos de clasificación son:

- Dado un conjunto de síntomas, conocer si un paciente tiene cierta enfermedad. Clase: *Sí* o *No*.
- Dadas unas características meteorológicas actuales, poder hacer la previsión del día siguiente. Clase: *Sol*, *Lluvia*, *Nubes*, *Granizo*, etcétera.

Tras entrenar el sistema, se obtendrá un modelo que servirá para clasificar nuevos ejemplos de acuerdo al conocimiento que haya sido capaz de aprender, prediciendo la clase a la que pertenecerá. Algunos modelos típicos en clasificación son: tablas de decisión, árboles de decisión, redes neuronales, sistemas de aprendizaje basado en instancias (IBL), etcétera.

Las técnicas de aprendizaje supervisado para clasificación se pueden aplicar a sistemas de planificación automática, siendo éste el objetivo del presente proyecto de investigación. La idea de combinar el aprendizaje supervisado con la planificación automática, se basa en emplear el conocimiento ofrecido por los modelos generados como soporte a la decisión del planificador. De esta manera, se pueden plantear dos opciones usando árboles de decisión:

1. Que el árbol se combine con las funciones heurísticas, ofreciendo información de los siguientes estados.
2. Que el planificador escoja directamente la acción indicada por el árbol.

En el siguiente apartado se entrará en detalle sobre qué es un árbol de decisión, qué elementos lo componen, para qué sirven y qué tipos de árboles de decisión podemos encontrar.

2.3.1.1. Árboles de decisión

En este apartado se va a hablar de los árboles de decisión dentro del entorno del aprendizaje automático, en relación con temas de clasificación. Hay que mencionar que este tipo de estructuras pueden ser empleadas en otros entornos que se salen de la temática de este proyecto.

Un árbol de decisión es una herramienta de soporte a la decisión que emplea un grafo en forma de árbol para representar gráficamente un modelo con el que se pretende predecir el valor de una variable, en base a los valores de otras variables (no tienen por qué ser todas las demás).

Como toda estructura en árbol, se compone de dos elementos: nodos y arcos (encargados de unir los nodos). A su vez, hay dos tipos de nodos:

- **Intermedio:** es un nodo que tiene uno o más descendientes. Contiene consultas (*queries*) sobre el valor de determinada variable.
- **Hoja:** es un nodo sin descendientes. Contiene el valor que se obtendría (predicción) para la variable objetivo, dados los valores de las variables de entrada indicados por el camino desde el nodo raíz (el nodo que no tiene padre) hasta ese nodo hoja.

Como ya se ha comentado, no es necesario que los nodos intermedios recojan todas las variables de un problema; sino que, en función de cómo haya sido el proceso de aprendizaje realizado, el árbol podrá basar sus predicciones en un conjunto de ellas, incluso en una sola.

En cuanto a los nodos hoja, hay que mencionar que, en algunas representaciones, el valor (que puede ser un número o una etiqueta de clase) para la variable objetivo se suele indicar junto a una probabilidad. Así, puede que por dos caminos diferentes se llegue a predecir la misma clase, pero la probabilidad de pertenencia a dicha clase sea mayor por una ruta que por otra.

La figura 2.1 muestra un ejemplo de árbol de decisión en el que se utilizan dos variables de entrada, el color de ojos del padre y el color de ojos de la madre, en función de los cuales se predice la variable objetivo: el color de ojos del hijo. En la figura se puede observar, además, lo mencionado en el párrafo anterior: el hijo tendrá un 75 % de tener los ojos marrones si ambos progenitores los tienen marrones, mientras que sólo tendrá un 50 % si la madre tiene los ojos verdes.

Existe un tipo de árboles de decisión que atañe a este proyecto, que son los árboles de decisión **relacionales**. Se distinguen de los anteriores (*proposicionales*) en dos cuestiones fundamentales:

1. Los nodos intermedios no hacen alusión a valores de atributos, sino a consultas lógicas sobre hechos que son ciertos en un estado determinado.
2. Estas consultas pueden compartir variables con nodos intermedios anteriores.

Ahora las consultas serán siempre binarias: valores *cierto* o *falso* en función de si se cumple o no la relación de variables indicada en la consulta.

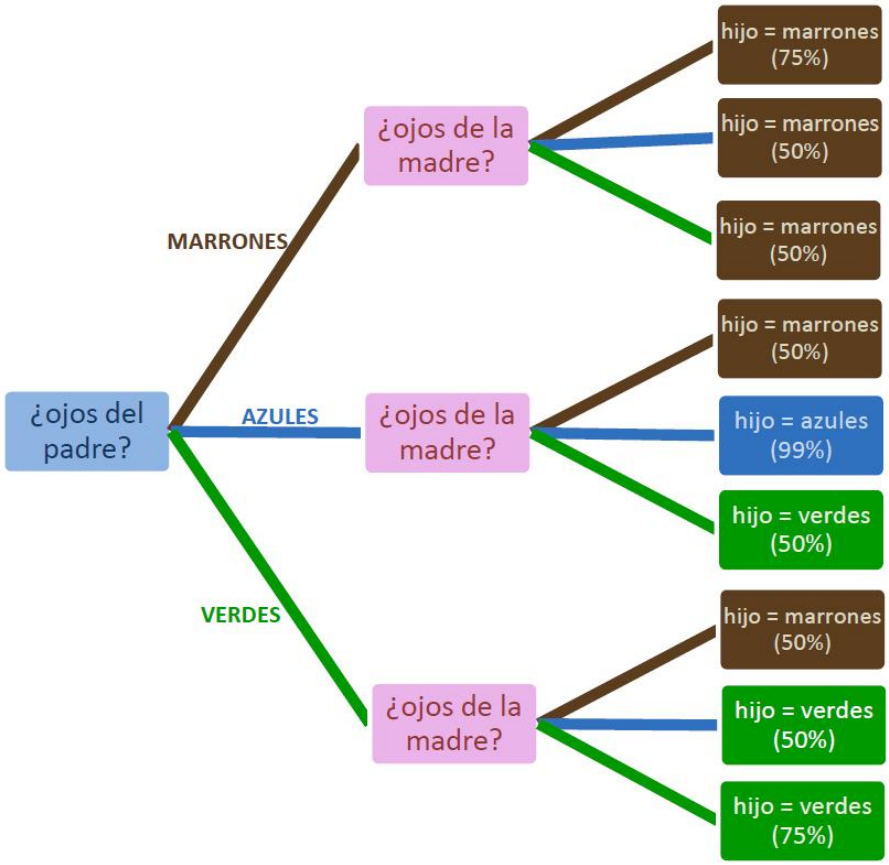

Figura 2.1: Árbol de decisión proposicional para predecir el color de ojos de un niño en función del color de ojos de sus padres.

La figura 2.2 muestra un ejemplo real de árbol relacional generado para el dominio *Satellite*. Los nodos intermedios, que terminan con un signo de interrogación, representan las consultas que han de ser evaluadas para poder ofrecer consejo sobre qué operador convendría escoger para un contexto determinado. Los nodos hoja muestran precisamente el operador seleccionado al llegar por cada camino. Además, estos nodos incluyen el número de ejemplos cubierto por esa regla. Por ejemplo, la acción *switch_ on* ha cubierto 44 ejemplos de un total de 59.

```
selected(-A,-B,-C)
helpful_calibrate(A,B,-D,-E,-F) ?
+--yes:[calibrate] 44.0 [[turn_to:0.0,switch_on:0.0,switch_off:0.0,
|                        calibrate:44.0,take_image:0.0]]
+--no:  helpful_take_image(A,B,-G,-H,-I,-J) ?
         +--yes:[take_image] 110.0 [[turn_to:0.0,switch_on:0.0,switch_off:0.0,
         |                          calibrate:0.0,take_image:110.0]]
         +--no:  helpful_switch_on(A,B,-K,-L) ?
                 +--yes:[switch_on] 59.0 [[turn_to:15.0,switch_on:44.0,
                 |                        switch_off:0.0,calibrate:0.0,
                 |                        take_image:0.0]]
                 +--no:  [turn_to] 149.0 [[turn_to:149.0,switch_on:0.0,
                                           switch_off:0.0,calibrate:0.0,
                                           take_image:0.0]]
```

Figura 2.2: Árbol de decisión relacional para predecir qué operador es más útil escoger dentro del dominio *Satellite*.

2.3.2. Aprendizaje mediante conjuntos de clasificadores

El aprendizaje supervisado se basa en buscar dentro de un espacio de hipótesis, aquéllas que sirvan para realizar buenas predicciones que permitan clasificar correctamente un problema determinado. La dificultad del proceso radica en ser capaces de encontrar buenas hipótesis dentro del espacio de búsqueda.

Para tratar de facilitar esto, existe el aprendizaje mediante conjuntos (*"ensemble learning"*). Se basa en combinar al mismo tiempo varias hipótesis con la esperanza de poder obtener una nueva, que sea mejor que todas ellas por separado. Para ello, es necesario que los diferentes conjuntos sean precisos y, a la vez, sean lo suficientemente distintos entre sí como para poder aportar información complementaria.

Por contra, al tener que evaluar un número mayor de hipótesis, el uso de conjuntos requiere (de forma general) más recursos que los que pueda emplear un sistema de aprendizaje simple.

Hay diferentes métodos aplicados al aprendizaje mediante conjuntos de clasificadores, como son bagging (Bootstrap AGGregatING), Boosting o Stacking.

- **Bagging:** dividir un conjunto de entrenamiento en varios, escogiendo los elementos mediante una selección aleatoria con reemplazo. Construir un clasificador con cada conjunto y generar salidas con cada clasificador para finalmente combinarlas, escogiendo el resultado mayoritario.

- **Boosting:** generar secuencialmente clasificadores, que tratan de resolver los problemas que no haya sido capaz de clasificar correctamente

el clasificador anterior.

- **Stacking:** construir diferentes clasificadores a partir de un conjunto inicial. En vez de combinar directamente sus salidas, éstas servirán de entrada (como atributos) a un nuevo clasificador (*metaclasificador*), que tomará la decisión final.

En el siguiente apartado se explicará el primero de ellos, por estar en relación con el proyecto realizado.

2.3.2.1. Bootstrap aggregating: Bagging

Bootstrap **Agg**regat**ing** (Bagging) (Breiman, 1996) es un meta-algoritmo diseñado para mejorar la estabilidad y precisión de los algoritmos de aprendizaje automático usados en clasificación (y regresión).

El funcionamiento de bagging consiste en generar varias versiones de un clasificador con distintos conjuntos de entrenamiento, cuyas salidas se combinan generando la idea de un único clasificador final.

Para ello, en primer lugar se actúa sobre el conjunto inicial de entrenamiento, que será la base sobre la que se creen los distintos clasificadores. Cada uno de ellos tendrá su propio conjunto de entrenamiento, generado a partir del inicial mediante una selección aleatoria con reemplazo. Este conjunto será del mismo tamaño que el inicial, pero ahora un mismo elemento del conjunto inicial puede estar varias veces en el conjunto final, de lo que deriva el hecho de que en los conjuntos de entrenamiento de los distintos clasificadores no estarán siempre todos los ejemplos disponibles. Esto aporta cierta diferenciación a la construcción de los clasificadores, que se espera que sea suficiente como para hacer rentable el uso de los mismos en lugar de un clasificador simple.

La tabla 2.1 muestra un ejemplo de cómo se ha dividido un conjunto de entrenamiento inicial en dos conjuntos de bagging. Se puede ver cómo todos ellos contienen el mismo número de elementos, pero los conjuntos de bagging contienen elementos repetidos y omiten algunos otros.

Después de construir los conjuntos de entrenamiento, se procede a crear cada uno de los clasificadores.

El siguiente paso es combinar las salidas de cada uno de ellos, para lo cual se pueden usar diferentes aproximaciones. Una de ellas, empleada normalmente por bagging, consiste en la implementación de un sistema de voto mayoritario, es decir, a la hora de predecir la salida de un ejemplo, se le otorgará la clase indicada por un mayor número de clasificadores dentro del conjunto utilizado.

Conjunto inicial	Conjunto Bagging-A	Conjunto Bagging-B
1	1	10
2	1	9
3	1	8
4	3	6
5	3	6
6	3	7
7	7	4
8	8	3
9	9	8
10	1	3

Tabla 2.1: Ejemplo de conjuntos de bagging.

De esta manera, mediante el uso de bagging se consiguen matizar las decisiones tomadas, ya que ahora la responsabilidad no cae exclusivamente en un clasificador, sino en la aportación de varios de ellos. Así, al emplearse las decisiones de la mayoría, se corrigen posibles desviaciones debidas al azar.

Además, dado que los conjuntos de entrenamiento empleados serán diferentes y, por tanto, los clasificadores también lo serán (en mayor o menor medida), se explorará un mayor espacio de búsqueda, haciendo que las soluciones propuestas sean más precisas que las que pudiera dar un sólo clasificador que se ha ceñido a un área concreta.

De todo lo comentado se puede extraer una conclusión, y es que el uso de bagging será especialmente atractivo para aquellos dominios en los que una ligera modificación del conjunto de entrenamiento inicial altere significativamente el clasificador construido.

Por último, es necesario mencionar que el funcionamiento de bagging se verá afectado por el número de conjuntos de entrenamiento (*"bags"*), pudiendo obtener resultados muy variados en función de la cantidad de grupos empleada y, por supuesto, afectando ésta al tiempo de entrenamiento.

2.4. TILDE y ACE

TILDE (Blockeel et al., 2000) (Blockeel y De Raedt, 1998) es un sistema de aprendizaje automático, que genera árboles de decisión relacionales. Fue desarrollado por H. Blockeel y L. De Raedt, y publicado en el año 1998 por la Universidad de Leiden, en los Países Bajos.

Surgió como una mejora del algoritmo C4.5 (Quinlan, 1993). Construye árboles de decisión que permiten predecir el valor de un atributo determinado, de acuerdo a cierta información de la base de datos que se esté usando.

TILDE ha sido usado por muchas herramientas de aprendizaje relacional. Una de ellas es ACE, desarrollada por parte del mismo equipo, que incluye TILDE además de otros algoritmos de aprendizaje, para los que sirve de interfaz común.

ACE emplea tres ficheros que recibe como entrada, todos ellos con el mismo nombre (el de la aplicación) pero diferente extensión:

1. **Fichero .kb:** contiene los ejemplos de entrenamiento y validación. *En el Apéndice B.3 puede consultarse un ejemplo de fichero de entrenamiento.*
2. **Fichero .bg *(opcional)*:** contiene la base de conocimiento del dominio.
3. **Fichero .s:** es el fichero de configuración. Permite ajustar ciertos parámetros de ACE. *En el Apéndice B.2 puede consultarse un ejemplo de fichero de configuración.*

2.5. ROLLER

ROLLER (de la Rosa et al., 2011) es un programa informático desarrollado por la UC3M, que genera planes automáticos empleando búsqueda heurística con la ayuda de árboles de decisión relacionales. Por un lado, emplea técnicas de planificación automática para generar los planes que resuelvan los problemas a los que se enfrente, siendo ésta su finalidad principal. Por otro, mediante el aprendizaje automático, se generan los árboles de decisión a partir de un entrenamiento realizado con ejemplos previos de planes anteriores, que servirán de soporte a la planificación.

Tras las pruebas realizadas, para diez dominios diferentes, se demostró que ROLLER obtenía buenos resultados en comparación con otros planificadores, FF y LAMA (primera solución encontrada).

Es un hecho que, para resolver problemas muy grandes, o bien cuando se están usando determinadas funciones heurísticas, engañosas respecto a la solución real, el número de nodos del árbol de búsqueda que hay que evaluar para generar el plan tiende a crecer. La idea de ROLLER es que, incluyendo en el planificador los árboles de decisión, se consigue reducir el número de nodos evaluados del árbol de búsqueda al tratar de generar un plan.

De esta manera, se podría entender la planificación heurística como una tarea de clasificación relacional, en la que el árbol de decisión ayude a escoger qué operador aplicar a partir de un estado. ROLLER escoge la mejor acción posible para diferentes contextos de un mismo dominio. Dichos contextos, denominados "contextos útiles" (*helpful contexts*), quedan definidos por:

1. Acciones útiles (*helpful actions*) del estado actual, definidas de forma heurística.

2. Metas aún no alcanzadas.

3. Predicados estáticos de la tarea de planificación.

El uso del contexto útil presenta varios aspectos positivos que justifican su utilización. En primer lugar, el conjunto de acciones útiles por lo general presenta una representación bastante compacta. En segundo lugar, además, el número de acciones útiles decrece (normalmente) a medida que van quedando menos metas por alcanzar. Por tanto, el proceso de unificación se vuelve más rápido a medida que el planificador se va acercando a su objetivo.

Por otro lado, el árbol de decisión, en su ayuda a la planificación, puede emplearse en ROLLER de dos formas:

1. Como una política de acción, escogiendo para cada estado la acción indicada por el clasificador. *Ésta será la estrategia empleada para este proyecto.*

2. Como una heurística para proporcionar información de los siguientes estados (*lookahead states*), usando el algoritmo de búsqueda del mejor primero (*Best First Search*) guiado por la heurística de FF.

A lo largo de esta sección se irán detallando los aspectos más relevantes del funcionamiento de ROLLER, con el fin de aclarar todo el proceso llevado a cabo por el sistema, incluyendo la creación del árbol de decisión y su uso posterior para tareas de planificación.

2.5.1. Representación del dominio

El dominio se representa mediante una formalización realizada con el lenguaje PPDL. Contiene:

- Una jerarquía de **tipos**.
- Un conjunto de **constantes**, que representan a los objetos que están presentes en todas las tareas del dominio. *Puede no haber ninguna constante para un dominio determinado.*
- Un conjunto de **predicados**.
- Un conjunto de **operadores**.

Un **problema de planificación** se compondrá de tres valores: las **constantes** propias del problema, su **estado inicial** y las **metas** que se quieren alcanzar.

En base a todo lo anterior, surgen nuevos conceptos:

- **Estado**: representa los predicados que son ciertos en un momento determinado.

- **Acción**: se corresponde con una instanciación de un operador, en el que cada variable del operador ha sido reemplazada por una constante (del mismo tipo).

Visto esto, se llega a la definición de **plan**, que es un conjunto de acciones que permiten, partiendo del estado inicial, satisfacer las metas propuestas.

En el apéndice B.1, puede verse un ejemplo de cómo es el fichero de dominio utilizado por ROLLER para el dominio *Blocksworld*.

2.5.2. Proceso de aprendizaje

ROLLER realiza un proceso de aprendizaje dividido en dos fases. En la primera de ellas, se construye un clasificador que permita escoger cuál es la **mejor acción** para diferentes contextos útiles (*operator classifier*). En la segunda fase se escoge, para cada operador del dominio, cuál es su **mejor instanciación** (*binding classifiers*). Para la construcción de los clasificadores de ambas fases se emplea el mismo conjunto de entrenamiento.

ROLLER genera los árboles mediante una herramienta externa, ACE (usando TILDE), que es la encargada de construir los clasificadores relacionales que se usarán más tarde para resolver tareas de planificación, aunque podrían usarse otras herramientas similares.

La figura 2.3 muestra la arquitectura de ROLLER durante el proceso de aprendizaje:

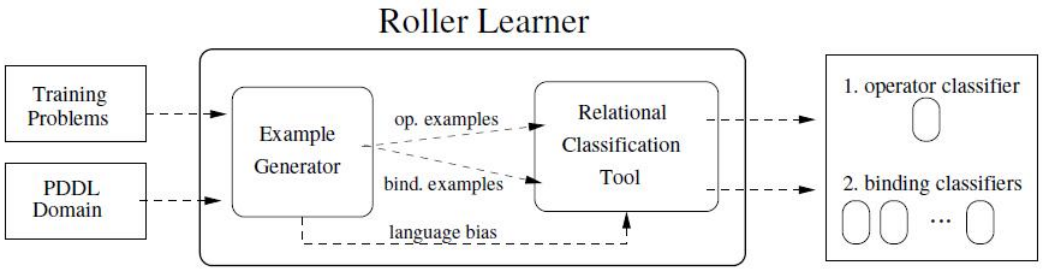

Figura 2.3: Arquitectura de ROLLER.

En los siguientes apartados se explicará en detalle cada una de las dos fases del aprendizaje de ROLLER.

2.5.2.1. Aprendizaje del árbol de operadores

Parar construir el árbol de operadores, se necesitan dos entradas:

1. **Ejemplos de entrenamiento**. Cada ejemplo se compone del operador seleccionado (que es la clase) junto al contexto útil en el que se selecciona. En detalle:
   - Clase: selected(*id-ejemplo*, *id-problema*, *clase-ejemplo*).
   - Predicados útiles: helpful_a(*parámetros-acción*), donde a es el operador instanciado (acción).
   - Predicados de las metas: target_goal_g(*parámetros-meta*), donde g es la meta.
   - Predicados estáticos: static_fact_p(*parámetros-predicado*), donde p es el predicado. Son fijos, compartidos para todos los ejemplos de un mismo problema.
2. **Restricciones del lenguaje**: restricciones sobre los argumentos de los predicados. No se asume ninguna restricción dependiente del dominio sino sobre otras cuestiones generales como los tipos de los argumentos de los predicados. Este conocimiento es extraído directamente de la definición PDDL del dominio.

La figura 2.4 muestra un ejemplo de entrenamiento para el dominio *Satellite*, en el que se ha seleccionado el operador *switch_on*.

```
% Example tr01_e1 from problem tr01
selected(tr01_e1,tr01,switch_on).
helpful_turn_to(tr01_e1,tr01,satellite0,groundstation1,star0).
helpful_turn_to(tr01_e1,tr01,satellite0,phenomenon2,star0).
helpful_turn_to(tr01_e1,tr01,satellite0,phenomenon3,star0).
helpful_turn_to(tr01_e1,tr01,satellite0,phenomenon4,star0).
helpful_switch_on(tr01_e1,tr01,instrument0,satellite0).
target_goal_have_image(tr01_e1,tr01,phenomenon3,infrared2).
target_goal_have_image(tr01_e1,tr01,phenomenon4,infrared2).
target_goal_have_image(tr01_e1,tr01,phenomenon2,spectrograph1).

% Static Predicates of problem
static_fact_calibration_target(tr01,instrument0,groundstation1).
static_fact_supports(tr01,instrument0,spectrograph1).
static_fact_supports(tr01,instrument0,infrared2).
static_fact_on_board(tr01,instrument0,satellite0).
```

Figura 2.4: Ejemplo de entrenamiento para el dominio *Satellite*.

La figura 2.5 muestra las restricciones para el mismo dominio. *rmode* indica los predicados que se pueden usar en el árbol, mientras que el *type* correspondiente detalla los tipos de los *rmode*.

```
% ---- The target concept ----
predict(selected(+IdExample,+IdProblem,-Operator)).
type(selected(idex,idprob,class)).
classes([turn_to,switch_on,switch_off,calibrate,take_image]).

% ---- The helpful context ----
% predicates for the helpful actions
rmode(helpful_turn_to(+IdExample,+IdProblem,+-S1,+-D1,+-D2)).
type(helpful_turn_to(idex,idprob,satellite,direction,direction)).

rmode(helpful_switch_on(+IdExample,+IdProblem,+-I1,+-S1)).
type(helpful_switch_on(idex,idprob,instrument,satellite)).

rmode(helpful_switch_off(+IdExample,+IdProblem,+-I1,+-S1)).
type(helpful_switch_off(idex,idprob,instrument,satellite)).

rmode(helpful_calibrate(+IdExample,+IdProblem,+-S1,+-I1,+-D1)).
type(helpful_calibrate(idex,idprob,satellite,instrument,direction)).

rmode(helpful_take_image(+IdExample,+IdProblem,+-S1,+-D1,+-I1,+-M1)).
type(helpful_take_image(idex,idprob,satellite,direction,instrument,mode)).

% predicates for the target goals
rmode(target_goal_pointing(+IdExample,+IdProblem,+-S1,+-D1)).
type(target_goal_pointing(idex,idprob,satellite,direction)).

rmode(target_goal_have_image(+IdExample,+IdProblem,+-D1,+-M1)).
type(target_goal_have_image(idex,idprob,direction,mode)).

% predicates for the static facts
rmode(static_fact_on_board(+IdProblem,+-I1,+-S1)).
type(static_fact_on_board(idprob,instrument,satellite)).

rmode(static_fact_supports(+IdProblem,+-I1,+-M1)).
type(static_fact_supports(idprob,instrument,mode)).

rmode(static_fact_calibration_target(+IdProblem,+-I1,+-D1)).
type(static_fact_calibration_target(idprob,instrument,direction)).
```

Figura 2.5: Ejemplo de restricciones del dominio *Satellite* para generar el árbol de operadores.

La figura 2.6 muestra un ejemplo de árbol de operadores generado para el dominio *Satellite*, que ya fue explicado en la sección 2.3.1.1.

```
selected(-A,-B,-C)
helpful_calibrate(A,B,-D,-E,-F) ?
+--yes:[calibrate] 44.0 [[turn_to:0.0,switch_on:0.0,switch_off:0.0,
|                         calibrate:44.0,take_image:0.0]]
+--no:  helpful_take_image(A,B,-G,-H,-I,-J) ?
         +--yes:[take_image] 110.0 [[turn_to:0.0,switch_on:0.0,switch_off:0.0,
         |                          calibrate:0.0,take_image:110.0]]
         +--no:  helpful_switch_on(A,B,-K,-L) ?
                 +--yes:[switch_on] 59.0 [[turn_to:15.0,switch_on:44.0,
                 |                        switch_off:0.0,calibrate:0.0,
                 |                        take_image:0.0]]
                 +--no:  [turn_to] 149.0 [[turn_to:149.0,switch_on:0.0,
                                           switch_off:0.0,calibrate:0.0,
                                           take_image:0.0]]
```

Figura 2.6: Árbol de decisión relacional para predecir qué operador es más útil escoger dentro del dominio *Satellite*.

2.5.2.2. Aprendizaje de los árboles de instanciaciones

Existirá un árbol de instanciaciones para cada uno de los operadores del dominio, indicando la mejor instanciación para cada uno de ellos.

Parar construir el árbol de instanciaciones se requieren dos entradas:

1. **Ejemplos de entrenamiento**: contiene los contextos útiles en los que el operador *"o"* fue seleccionado, junto con sus posibles instanciaciones (incluyendo acciones útiles (*helpful*) o no-útiles (*non-helpful*)). Las instanciaciones se representan mediante el predicado "selected_*o*". La representación del contexto útil se hace de la misma manera que para el árbol de operadores.

2. **Restricciones del lenguaje**: tiene la misma forma que durante el aprendizaje del árbol de operadores, con la diferencia de que ahora se incluye el predicado "selected_*o*" para indicar el operador.

La figura 2.7 muestra un ejemplo de entrenamiento para el dominio *Satellite* y la acción *switch_on*.

```
% Example tr07_e63 from problem tr07
selected_switch_on(tr07_e63,tr07,instrument0,satellite0,rejected).
selected_switch_on(tr07_e63,tr07,instrument1,satellite0,selected).
helpful_switch_on(tr07_e63,tr07,instrument0,satellite0).
helpful_switch_on(tr07_e63,tr07,instrument1,satellite0).
helpful_turn_to(tr07_e63,tr07,satellite0,star1,star2).
helpful_turn_to(tr07_e63,tr07,satellite0,star5,star2).
helpful_turn_to(tr07_e63,tr07,satellite0,phenomenon7,star2).
helpful_turn_to(tr07_e63,tr07,satellite0,phenomenon8,star2).
target_goal_have_image(tr07_e63,tr07,phenomenon8,spectrograph2).
target_goal_have_image(tr07_e63,tr07,phenomenon7,spectrograph2).
target_goal_have_image(tr07_e63,tr07,star5,image1).

% Static Predicates of problem
static_fact_calibration_target(tr07,instrument0,star1).
static_fact_calibration_target(tr07,instrument1,star1).
static_fact_supports(tr07,instrument0,image1).
static_fact_supports(tr07,instrument1,spectrograph2).
static_fact_supports(tr07,instrument1,image1).
static_fact_supports(tr07,instrument1,image4).
static_fact_on_board(tr07,instrument0,satellite0).
static_fact_on_board(tr07,instrument1,satellite0).
```

Figura 2.7: Ejemplo de entrenamiento para el dominio *Satellite*.

La figura 2.8 muestra las restricciones. Se muestra sólo la parte que difiere con el fichero empleado para generar el clasificador de operadores.

```
% ---- The target concept ----
predict(selected_switch_on(+IdExample,+IdProblem,+INST0,+SAT1,-Class)).
type(selected_switch_on(idex,idprob,instrument,satellite,class)).
classes([selected,rejected]).

% ---- The helpful context ----, the same as in the operator classification
...
```

Figura 2.8: Ejemplo de restricciones del dominio *Satellite* para generar el árbol de instanciaciones de la acción *switch_ on*.

La figura 2.9 muestra un ejemplo de árbol de instanciaciones generado para la acción *switch_ on* del dominio *Satellite*.

```
selected_switch_on(-A,-B,-C,-D,-E)
helpful_switch_on(A,B,C,D) ?
+--yes:  [selected] 249.0 [[selected:213.0,rejected:36.0]]
+--no:  [rejected] 63.0 [[selected:2.0,rejected:61.0]]
```

Figura 2.9: Árbol de instanciaciones para la acción *switch_on* del dominio *Satellite*.

2.5.3. Generación de ejemplos de entrenamiento

Los ejemplos de entrenamiento utilizados por ROLLER son instancias de decisiones tomadas al resolver problemas de entrenamiento. Para su creación, se hace un proceso dividido en tres fases:

1. Generación de soluciones.
2. Selección de mejores soluciones.
3. Extracción de ejemplos a partir de las soluciones seleccionadas.

La **generación de soluciones** se realiza mediante el algoritmo *Best-First Branch and Bound* (BFS-BnB), con el que se resuelve cada uno de los problemas de entrenamiento, generando múltiples soluciones. Si el espacio de búsqueda no se ha explorado lo suficiente en un tiempo establecido, se descarta el problema y no se generan ejemplos a partir de él. Es necesario, por tanto, que los problemas sean lo suficientemente pequeños. Además, deberán ser representativos, para que ROLLER pueda extraer conocimiento relevante, que le permitirá planificar mejor próximos problemas para ese dominio.

Al final de la búsqueda, el algoritmo devuelve el conjunto de soluciones de mejor coste. De ellas, ROLLER escoge algunas que serán usadas para generar el conjunto de entrenamiento. La elección se lleva a cabo dando prioridad a las acciones que:

- Generan **más alternativas**, caminos diferentes a la solución.
- Alcanzan las **metas "más difíciles"**. Se consideran más difíciles las metas que sólo se pueden alcanzar mediante una acción, frente a las que se pueden alcanzar mediante varias.
    - Por ejemplo, en el dominio *Satellite*, se considera que es más difícil la meta *"instrumento encendido"* (se logra sólo con la acción *"encender instrumento I"*) frente a *"apuntar a una dirección"* (que se obtiene con distintas instancias del operador *"girar hacia dirección D1 desde dirección D2"*).

Tras seleccionar las mejores soluciones, se procederá a extraer de ellas los ejemplos de entrenamiento. Los ficheros generados, conteniendo los ejemplos, serán los que sirvan de entrada a ACE para construir los árboles de decisión. Ejemplos de ficheros de entrenamiento ya han sido mostrados en las figuras 2.4 (para el árbol de operadores) y 2.7 (para el árbol de instanciaciones).

2.5.4. Planificación con árboles relacionales

En esta sección se va a explicar cómo utilizar el conocimiento adquirido en las fases anteriores para mejorar sistemas de planificación heurística. Se detallará cómo se realiza la ordenación de las acciones y cómo se utiliza posteriormente para generar planes utilizando esto como política de acción.

2.5.4.1. Ordenación de acciones

Para cada estado, existirá un conjunto de acciones que se puedan aplicar. La ordenación de estas acciones se realizará mediante un proceso de **unificación** (*matching*) de cada una de ellas con el árbol de operadores y, posteriormente, con el correspondiente árbol de instanciaciones.

El algoritmo divide el conjunto de acciones aplicables en dos: útiles y no-útiles. Después, unifica el contexto útil con el árbol de operadores, lo que proporciona un nodo hoja que contiene una lista con todos los operadores, ordenados por el número de ejemplos cubierto por cada uno de ellos durante la fase de entrenamiento. Con este número, se inicializa el valor de *prioridad* de cada acción útil, seleccionando sólo aquéllas que tengan un valor mínimo de 1.

Después, para cada una de las acciones seleccionadas, se hace el proceso de "matching" con el árbol de instancias correspondiente. El nodo hoja resultante, devolverá esta vez dos valores: el número de veces que la acción "a" fue seleccionada, y el número de veces que fue rechazada durante la fase de entrenamiento. Con estos valores se calculará el ratio de selección:

$$ratio\_de\_seleccion(a) = \frac{seleccionada(a)}{seleccionada(a) + rechazada(a)}$$

Cuando el denominador es cero, se asume que el valor del ratio también lo es.

Una vez calculado el ratio de selección para cada acción, se actualiza el valor de prioridad para cada una sumando el ratio al valor anterior de prioridad.

La figura 2.10 muestra el psuedo-código del algoritmo de ordenación.

DT-Filter-Sort (A,$\mathcal{H}$,T): *lista ordenada de acciones aplicables*

A: Lista de acciones
$\mathcal{H}$: Contexto Útil
T: Árboles de Decisión

---

acciones-seleccionadas = $\emptyset$
HA = acciones-útiles (A, $\mathcal{H}$)
NON-HA = $A \setminus HA$
nodo-hoja = clasificar-árbol-operadores (T, $\mathcal{H}$)
for each a **in** *HA* **do**
    prioridad(a) = valor-operador-nodo-hoja(*nodo-hoja*, a)
    if *prioridad(a)* > 0 **then**
        (seleccionada(a),rechazada(a)) = clasificar-árbol-instanciaciones (T, $\mathcal{H}$, a)

        $ratio\_de\_selección(a) = \frac{seleccionada(a)}{seleccionada(a)+rechazada(a)}$

        $prioridad(a) = prioridad(a) + ratio\_de\_selección(a)$
        $acciones\text{-}seleccionadas = acciones\text{-}seleccionadas \cup \{a\}$

max-HA-prioridad = $\max_{a \in acciones\text{-}seleccionadas} prioridad(a)$
for each a **in** *NON-HA* **do**
    prioridad(a) = valor-operador-nodo-hoja (*nodo-hoja*,a)
    if *prioridad(a)* $>$ *max-HA-prioridad* **then**
        (seleccionada(a),rechazada(a)) = clasificar-árbol-instanciaciones (T, $\mathcal{H}$, a)

        $ratio\_de\_selección(a) = \frac{seleccionada(a)}{seleccionada(a)+rechazada(a)}$

        $prioridad(a) = prioridad(a) + ratio\_de\_selección(a)$
        $acciones\text{-}seleccionadas = acciones\text{-}seleccionadas \cup \{a\}$

return ordenar (*acciones-seleccionadas*, *prioridad*)

Figura 2.10: Pseudo-código del algoritmo de ordenación.

La prioridad de las acciones no-útiles se calcula de una forma similar, sólo que ahora únicamente se consideran acciones cuya prioridad inicial es mayor que la prioridad máxima de las acciones útiles.

Finalmente, las acciones seleccionadas se ordenan de mayor a menor valor de prioridad. La lista ordenada será la salida del algoritmo.

2.5.4.2. Algoritmo H-Context Policy

El algoritmo "helpful context-action policy", cuyo pseudo-código se muestra en la figura 2.11, realiza una búsqueda hacia delante, aplicando en cada estado la mejor acción posible.

Para ello, emplea una lista abierta, que contiene los nodos que han de ser expandidos, que se extraen en orden. Después de extraer un nodo (y no al meterlo en la lista), se evalúa con la heurística de FF. Esta evaluación proporciona un valor heurístico "h" y el conjunto de acciones útiles "HA" (*Helpful Actions*), necesarias para construir el contexto útil. Por su parte, el valor heurístico se emplea para dos tareas: continuar la búsqueda cuando se alcanza un "callejón sin salida" ($h = \infty$) o saber cuándo se alcanzado una meta (h=0).

Con el contexto útil, el conjunto de acciones aplicables desde el nodo, y los árboles de decisión, se llama al algoritmo de ordenación explicado en el apartado anterior, que devolverá un conjunto ordenado de acciones aplicables. Se generarán los sucesores del nodo usando estas acciones y éstos se introducirán al principio de la lista abierta, de manera que se garantice el orden. Hay que mencionar que el conjunto devuelto por el algoritmo de ordenación podrá tener menos elementos de los que se le pasan como entrada, ya que algunos habrán sido rechazados. Las acciones rechazadas, se meterán entonces en otra lista (*delayed-list*). Esta última, se usará en el caso de que, en un momento dado, la lista abierta se haya quedado vacía, en cuyo caso se tomará el primer elemento de la lista de rechazados y se meterá en la lista abierta. A partir de ese momento, se empleará de nuevo únicamente la lista abierta.

Con este algoritmo, cada nodo mantiene un puntero al padre, de manera que se pueda localizar el camino una vez que se ha encontrado la solución. Además, para cada nodo se almacena un valor "g" que representa la longitud de la ruta desde dicho nodo hasta el nodo inicial.

Hay que mencionar también, que en la lista abierta sólo se introducen nodos que cumplan dos características:

- Que no sean estados repetidos.
- Que sean estados repetidos con un menor valor "g" que el anterior.

El resto de nodos son podados, de manera que se garantice la solución más corta, encontrada hasta el momento, para cada estado.

Depth-First H-Context Policy (I, G, T): *plan*

I: Estado Inicial
G: Metas
T: Árboles de Decisión

---

lista-abierta $= \{I\}$;
lista-rechazados $= \emptyset$;
while *lista-abierta* $\neq \emptyset$ **do**
    $n =$ extraer (*lista-abierta*)
    $(h, HA) =$ evaluar (n, G) */*calcular heurística FF*/*
    if $h = \infty$ **then** */*encontrado callejón sin salida*/*
        continue
    if $h = 0$ **then** */*estado meta*/*
        return camino(I, n)
    $\mathcal{H} =$ contexto-útil (HA, G, n)
    $AA =$ acciones-aplicables (n)
    $AA' =$ DT-Filter-Sort $(AA, \mathcal{H}, T)$ */*ordenar y filtrar acciones*/*
    candidatos = generar-sucesores (n, AA')
    lista-abierta = meter-ordenados-lista-abierta (*candidatos*, *lista-abierta*)
    candidatos-rechazados = generar-sucesores $(n, AA \setminus AA')$
    lista-rechazados = meter-ordenados-lista (*candidatos-rechazados*,
                        lista-rechazados)
    while *lista-abierta* $= \emptyset$ **and** *lista-rechazados* $\neq \emptyset$ **do**
        lista-abierta = { extraer (*lista-rechazados*) }
return fallo

Figura 2.11: Pseudo-código del algoritmo DFHCP.

Capítulo 3

Desarrollo del proyecto

Resumen: En este capítulo se va a explicar la idea principal del proyecto desarrollado y qué se ha hecho para llevarla a cabo. Se explicarán los nuevos algoritmos implementados en ROLLER, indicando las diferencias respecto al funcionamiento 'clásico", visto en el capítulo de "Estado de la cuestión". Más tarde, se comentarán las partes fundamentales del sistema realizado, indicando dónde encaja lo anterior, y se detallará un esquema de alto nivel de la secuencia de pasos seguida durante los procesos de aprendizaje y planificación. Finalmente, se explicarán los nuevos módulos implementados para automatizar dichos procesos.

3.1. Aproximación de bagging en ROLLER

La idea principal de este proyecto es ampliar un sistema ya existente, ROLLER, para que sea capaz de generar planes empleando no sólo un árbol de decisión, sino varios a la vez, combinados mediante técnicas de bagging.

El sistema ROLLER es, por tanto, la base del proyecto. Ya se encontraba implementado previamente y, para este nuevo sistema, ha sido ampliado en algunas de sus tareas, de manera que se permita la inclusión de metaclasificadores.

El objetivo fundamental es implementar distintas aproximaciones que sean capaces de generar planes de manera automática utilizando varios árboles de decisión relacionales al mismo tiempo. Para ello, ha sido necesario, además de crear los algoritmos pertinentes, modificar determinadas estructuras manejadas por ROLLER.

Las dos políticas diseñadas son:

- Agregación de la información de todos los árboles: algoritmo TBAP.

- Selección de la información de los árboles de manera alternativa, usando un árbol cada vez: algoritmo MQTBP.

Ambos algoritmos han sido diseñados empleando como partida el algoritmo DFHCP (Depth-First H-Context Policy), explicado en detalle en el Apartado 2.5.4.2.

A continuación se comentan los nuevos algoritmos implementados.

3.1.1. Tree-Bagging Aggregation Policy (TBAP)

El algoritmo TBAP (Tree-Bagging Aggregation Policy) se corresponde con el que se ha llamado "de agregación". Se basa en combinar de forma numérica la información proporcionada por todos los árboles cargados en memoria. De esta manera, cuantos más árboles haya, mayor será la información manejada por el planificador a la hora de escoger los operadores; ya que, para cada una de las decisiones que tome, tendrá en cuenta todas las enseñanzas aportadas por cada uno de los árboles, de forma combinada.

Para ello, al realizar el proceso de unificación, se suman las decisiones proporcionadas por cada uno de los árboles de decisión disponibles. Esto conlleva que este proceso ahora tenga que evaluar la utilidad de los operadores para un contexto por cada uno de los árboles, para finalmente combinar la información.

El algoritmo como tal mantiene la misma idea empleada por el DFHCP, ya que ahora el TBAP tendrá también una única lista abierta, de la que irá extrayendo nodos hasta dar con una solución, o hasta vaciar la lista. La diferencia está en que ahora ha sido necesario modificar el algoritmo de ordenación para sumar los valores de las salidas.

La figura 3.1 muestra el pseudo-código del algoritmo de ordenación para el caso de agregación. Entrando más en detalle, el algoritmo de ordenación para el proceso de agregación (*DT-Filter-Sort-Aggregation*) tiene ahora las siguientes particularidades:

- En primer lugar, la lista de acciones seleccionadas será ahora una matriz, donde cada fila se corresponderá a la lista de acciones seleccionadas para cada uno de los grupos (bags).

- Posteriormente, se rellena la matriz, generando para cada grupo sus listas de prioridades, acciones seleccionadas y acciones rechazadas, en base a su propio árbol de decisión. De manera evidente, ahora este proceso multiplicará el tiempo dedicado en relación al número de agrupaciones empleado.

DT-Filter-Sort-Aggregation (A,$\mathcal{H}$,T,B): *lista ordenada de acciones aplicables*

A: Lista de acciones
$\mathcal{H}$: Contexto Útil
T: Vector de Árboles de Decisión para cada bag
b, tal que $0 < b < B$

HA = acciones-útiles (A, $\mathcal{H}$)
NON-HA = $A \setminus$ *HA*
*/*Para cada bag, se realiza todo el proceso y se mete su lista en la matriz*/*
for each *bag b* **do**
    nodo-hoja = clasificar-árbol-operadores (*T[b]*, $\mathcal{H}$)
    for each *a* **in** *HA* **do**
        prioridad[b][a] = valor-operador-nodo-hoja(*nodo-hoja*, *a*)
        if *prioridad[b][a]* > 0 **then**
            (seleccionada[b][a],rechazada[b][a]) = clasificar-árbol-instanciaciones(*T[b]*,$\mathcal{H}$, *a*)
    for each *a* **in** *NON-HA* **do**
        prioridad[b][a] = valor-operador-nodo-hoja (*nodo-hoja*,*a*)
        if *prioridad[b][a]* > 0 **then**
            (seleccionada[b][a],rechazada[b][a]) = clasificar-árbol-instanciaciones(*T[b]*,$\mathcal{H}$, *a*)
*/*Inicialización de variables de valores agregados*/*
for each *a* **in** *HA* $\cup$ *NON-HA* **do**
    prioridad(a) = 0
    seleccionada(a) = 0
    rechazada(a) = 0
*/*Agregación de valores*/*
for each *bag b* **do**
    for each *a* **in** *HA* $\cup$ *NON-HA* **do**
        prioridad(a) = *prioridad(a)* + *prioridad[b][a]*
        seleccionada(a) = *seleccionada(a)* + *seleccionada[b][a]*
        rechazada(a) = *rechazada(a)* + *rechazada[b][a]*
*/*Selección de acciones con los valores agregados*/*
for each *a* **in** *HA* **do**
    if *prioridad(a)* > 0 **then**
        $ratio\_de\_selección(a) = \frac{seleccionada(a)}{seleccionada(a)+rechazada(a)}$
        prioridad(a) = *prioridad(a)* + *ratio_de_selección(a)*
        acciones-seleccionadas = *acciones-seleccionadas* $\cup\{a\}$
max-HA-prioridad = $\max_{a \in acciones\text{-}seleccionadas} prioridad(a)$
for each *a* **in** *NON-HA* **do**
    if *prioridad(a)* $>$ *max-HA-prioridad* **then**
        $ratio\_de\_selección(a) = \frac{seleccionada(a)}{seleccionada(a)+rechazada(a)}$
        prioridad(a) = *prioridad(a)* + *ratio_de_selección(a)*
        acciones-seleccionadas = *acciones-seleccionadas* $\cup\{a\}$
return ordenar (*acciones-seleccionadas*, *prioridad*)

Figura 3.1: Pseudo-código del algoritmo de ordenación para agregación de valores (TBAP).

Otra de las diferencias que presenta el algoritmo de ordenación para el proceso de agregación, es que ha de unir la información de todos los grupos en uno solo. Para ello utiliza nuevos vectores, cuyos valores serán formados mediante la suma de los valores correspondientes de cada uno de los grupos. Así, por ejemplo, el número de veces que una acción ha sido seleccionada, se obtendrá como la suma de veces que ha sido seleccionada en cada uno de los árboles disponibles. El ratio de selección de una acción se calculará con la fórmula:

$$ratio\_de\_seleccion(a) = \frac{\sum_{b=1}^{B} seleccionada[b][a]}{\sum_{b=1}^{B} seleccionada[b][a] + \sum_{b=1}^{B} rechazada[b][a]}$$

El último paso novedoso es que ahora es necesario hacer una pasada más para comprobar las acciones seleccionadas (útiles y no-útiles) de la lista de valores agregados.

Posteriormente, se realizará la ordenación de esta última lista de igual manera que con el algoritmo de ordenación básico.

A partir de ese momento, el funcionamiento del algoritmo TBAP será idéntico al de DFHCP, ya que ambos actuarán sobre un único contexto, abstrayéndose el sistema del hecho de que en realidad el contexto de TBAP haya sido formado por la combinación de varios clasificadores.

3.1.2. Multiple Queue Tree-Bagging Policy (MQTBP)

El algoritmo MQTBP (Multiple Queue Tree Bagging Policy) emplea "múltiples listas" simultáneamente. El proceso de unificación y ordenación, en este caso, es idéntico al realizado por el algoritmo inicial, el DFHCP, con la diferencia de que ahora es necesario hacerlo una vez por cada árbol utilizado.

Para tomar las decisiones en cada paso, el planificador escoge de forma alternativa la información de los árboles de los que expandir un nodo. Esto quiere decir que si, por ejemplo, se tienen tres árboles, la primera decisión se realizará usando el primer árbol de decisión; la siguiente, el segundo; después, el tercero y, de nuevo, volverá al primero. De esta forma, este algoritmo hace uso de todos los árboles de decisión generados, pero no a la vez, a diferencia de lo que ocurría con el algoritmo TBAP.

La figura 3.2 muestra el pseudo-código del algoritmo MQTBP.

Multiple Queue Tree-Bagging Policy (I, G, T, B): *plan*

I: Estado Inicial
G: Metas
T: Vector de Árboles de Decisión para cada bag
B: Número de Bags

```
/*matriz de listas abiertas para cada bag*/
lista-abierta = [][]
/*matriz de listas de rechazados para cada bag*/
lista-rechazados = [][]
/*inicialización*/
for b < B do
    lista-abierta[b] = {I};
    lista-rechazados[b] = ∅;
    b = b + 1
/*contadores para saber qué lista escoger*/
contador = 0
k = 0
/*mientras no estén vacías todas las listas*/
while lista-abierta ≠ ∅ do
    /*se selecciona la lista correspondiente, con la operación módulo*/
    k = contador % B
    n = extraer (lista-abierta[k])
    if ya-expandido(n) = TRUE then
        continue
    (h, HA) = evaluar (n, G)   /*calcular heurística FF*/
    if h = ∞ then   /*encontrado callejón sin salida*/
        continue
    if h = 0 then   /*estado meta*/
        return camino(I, n)
    H = contexto-útil (HA, G, n)
    AA = acciones-aplicables (n)
    for b < B do
        AA' = DT-Filter-Sort (AA, H, T[b]) /*ordenar y filtrar acciones*/
        candidatos = generar-sucesores (n, AA')
        lista-abierta[b] = meter-ordenados-lista-abierta (candidatos,
                           lista-abierta[b])
        candidatos-rechazados = generar-sucesores (n, AA \ AA')
        lista-rechazados[b] = meter-ordenados-lista (candidatos-rechazados,
                              lista-rechazados[b])
        while lista-abierta[b] = ∅ and lista-rechazados[b] ≠ ∅ do
            lista-abierta[b] = { extraer (lista-rechazados[b]) }
        b = b + 1
    contador = contador + 1
return fallo
```

Figura 3.2: Pseudo-código del algoritmo de múltiples listas (MQTBP).

A continuación se entrará en detalle sobre los aspectos más relevantes de todo el proceso.

En primer lugar, como ya se ha comentado, la primera diferencia respecto a los algoritmos anteriores es que ahora no se contará con una única lista abierta, de la que ir extrayendo los nodos. Sino que el algoritmo MQTBP contará con una lista de este estilo por cada árbol de decisión generado.

Para ello, en una misma llamada al algoritmo MQTBP, se realizará el proceso de unificación para todos los grupos, actualizando así cada uno su propia lista de nodos.

De nuevo, al igual que ocurría con el algoritmo TBAP, esta idea conlleva un incremento del tiempo de unificación respecto a DFHCP, ya que ahora, en cada llamada al algoritmo, hay que hacer la unificación tantas veces como grupos se estén usando en la búsqueda.

La segunda diferencia significativa respecto a los demás algoritmos es la manera de combinar la información de todos los árboles de decisión construidos. En este caso, en vez de sumar los valores aportados por cada uno de los grupos y unirlos en una lista, lo que se hace es ir escogiendo una de las listas cada vez.

Para garantizar que se consultan todos los árboles creados, la elección de la lista se realiza de manera secuencial, empezando siempre por la misma, la lista correspondiente al primer grupo, siguiendo por la segunda, etcétera, y extrayendo cada vez un nodo de cada lista elegida.

Una diferencia respecto al algoritmo DFHCP es que ahora, al extraer un nodo de una lista y generar sus sucesores, estos últimos tendrán que meterse en todas las listas. En la lista correspondiente al árbol que se ha escogido en esa iteración, se meterán al inicio (como ya ocurría con DFHCP), ya que son los sucesores del primer nodo (el que se ha extraído). En el resto de listas, se introducirá en el lugar ocupado por ese nodo, de manera que se preserve el orden de cada lista.

El algoritmo se terminará cuando se haya obtenido una solución o cuando todas las listas estén vacías.

3.2. Arquitectura general del sistema

El sistema está formado por tres módulos principales. Dos de ellos (*ROLLER y TILDE-ACE*), se encargan de realizar diversas tareas para poder generar un plan que sea capaz de resolver un problema. A su vez, estos dos módulos son usados por el tercero de ellos (*Experimenter*), que sirve de nexo entre los anteriores, automatizando todas las pruebas para varios dominios, problemas y casos.

Los módulos principales son:

1. Subsistema 1 (*Experimenter*): consiste en varios programas que se en-

cargan de automatizar las pruebas, realizando las llamadas pertinentes a los otros dos subsistemas. Además genera, de forma previa a estas llamadas, la estructura de ficheros y directorios necesaria para la correcta creación de los múltiples árboles de decisión. Supone el punto de partida de la ejecución del sistema.

2. Subsistema 2 (*ROLLER*): ampliado en sus funciones para permitir el manejo de varios árboles de decisión de manera simultánea (empleando técnicas de bagging), siendo esta la finalidad principal del proyecto. Se emplea para generar los ficheros de entrenamiento y para resolver los problemas en la etapa de validación.

3. Subsistema 3 (*TILDE-ACE*): se emplea para generar los árboles de decisión. Este módulo no ha sido desarrollado para este proyecto, simplemente es utilizado por ofrecer funcionalidades requeridas para el sistema que no era necesario programar de nuevo. Se emplea "tal cual", por lo que no se ha creído conveniente entrar aquí en detalle sobre su funcionamiento, estando éste detallado en fuentes oficiales como (Blockeel et al., 2000) y (Blockeel y De Raedt, 1998).

En la figura 3.3 puede verse una representación de la secuencia seguida para un dominio, desde que se inicia el sistema, pasando por la creación de estructuras necesarias, el entrenamiento y generación de los árboles de decisión y, finalmente, el proceso de validación y de obtención de resultados.

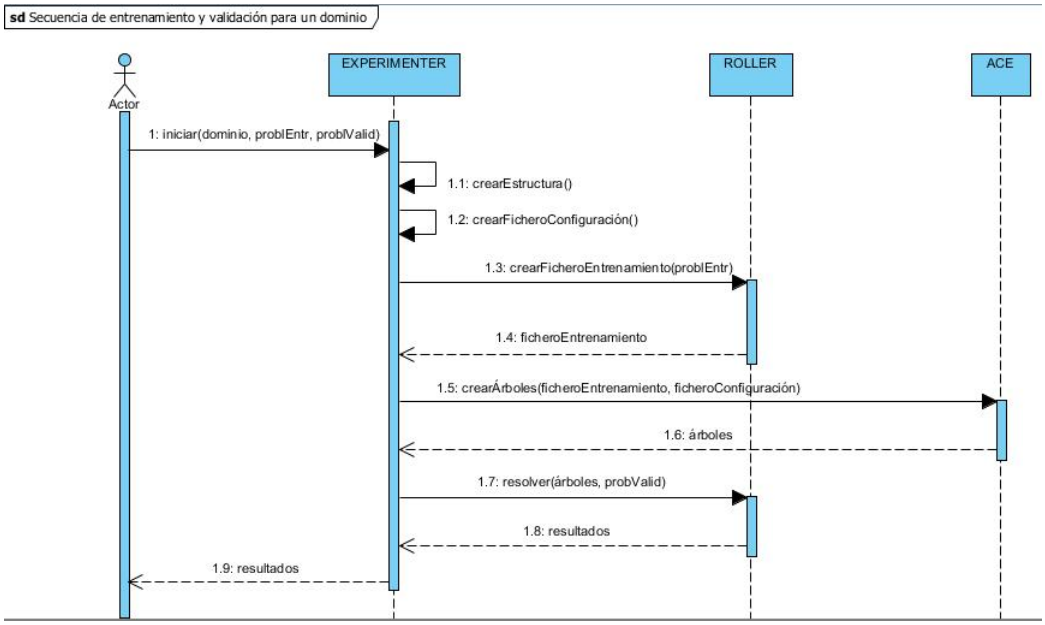

Figura 3.3: Diagrama de secuencia del proceso de entrenamiento y validación.

3.3. Descripción del módulo Experimenter

En esta sección se detallará el módulo Experimenter, indicando las diferentes partes de qué se compone y cuál es la finalidad de cada una de ellas. Se dará una visión cercana al código implementado, explicando las entradas y salidas y el proceso de funcionamiento.

3.3.1. Entrenador

El programa entrenador (roller_trainer.py) es un programa implementado en Python, formado por unas 570 líneas de código, que tiene por finalidad generar árboles de decisión para varios grupos (o bags).

Para ello, una vez que se ha decidido cuántos conjuntos va a haber (el número de bags es un parámetro), el programa entrenador construirá la estructura de directorios necesaria para cada uno de ellos. *(El detalle de la estructura requerida se muestra en la sección A.3 del Apéndice A).*

Después, a cada grupo se le asignará un conjunto de problemas de entrenamiento con el que se generará su propio clasificador (árbol de decisión en este caso). El conjunto de problemas para cada grupo se obtendrá a partir del conjunto de entrenamiento inicial para el dominio con el que se esté trabajando. Cada uno de los conjuntos finales tendrá exactamente el mismo número de problemas que el conjunto inicial. Lo que distingue a cada uno de ellos es que sus elementos se escogen a partir del conjunto inicial mediante una selección con reemplazo: esto es, se selecciona un elemento del conjunto inicial al azar, se copia al conjunto final, pero no se elimina del conjunto inicial; lo que en la práctica supone que un mismo elemento puede ser seleccionado más de una vez.

El siguiente paso llevado a cabo por el programa entrenador será generar una serie de ficheros intermedios (de configuración y de entrenamiento), necesarios para el correcto funcionamiento del sistema, ya que servirán de entrada al módulo ACE. El fichero de configuración será construido por el entrenador, mientras que el fichero de entrenamiento será generado por ROLLER.

Posteriormente, por tanto, el programa entrenador hará las llamadas pertinentes a ACE para que genere los árboles de decisión para cada uno de los conjuntos de entrenamiento, almacenando cada uno de ellos dentro de la estructura de directorios creada anteriormente.

A continuación se explica todo esto más en detalle.

En primer lugar, las entradas recibidas por este programa son:

1. **Fichero de dominio:** contiene la definición del dominio, esto es, sus tipos de elementos, predicados y acciones, en lenguaje PDDL (Planning Domain Definition Language). *(El Apéndice B.1 contiene un ejemplo completo de fichero de dominio.)*

2. **Nombre de la carpeta de destino:** en la cual se generará la estructura de directorios, conteniendo los árboles que se creen.

3. **Número de grupos:** número de grupos (bags) que se crearán para esa ejecución del programa.

4. **Prefijo de los problemas de entrenamiento:** en un conjunto de problemas de entrenamiento, cada miembro comenzará por el mismo prefijo, siendo la parte final la que distingue un problema de otro dentro del mismo grupo. Por ejemplo, los problemas: j2r-train01-8-001 y j2r-train01-8-002.

5. **Tiempo máximo:** tiempo medido en segundos, que como máximo se puede emplear al tratar de resolver un problema durante el entrenamiento.

6. **Flag de depuración:** valor binario para indicar si se desea mostrar o no una traza indicativa del progreso del programa durante la ejecución.

A la vista de lo anterior, hay que recordar que, de forma previa a la ejecución del programa, es necesario contar con el fichero PDDL del dominio y con un conjunto de problemas de entrenamiento para ese dominio, donde cada problema estará modelado empleando también el lenguaje PDDL. Todos los problemas del conjunto se encontrarán en un mismo directorio, y cada problema estará en un fichero distinto, donde su prefijo será igual para todos los problemas del mismo conjunto.

La ejecución del programa sigue los siguientes pasos:

1. Lectura de parámetros de entrada.

2. Extracción de información del dominio (nombre, acciones, predicados, etcétera).

3. Creación de la estructura de carpetas. En caso de existir, se elimina completamente tanto el contenido como la propia estructura y se crea desde cero. Dentro de la carpeta principal, se crea una carpeta baggingX (donde X es el número de grupos) por cada grupo. En caso de que el número de grupos sea 1, este paso se omite (esto quiere decir que esta prueba no usará bagging). Dentro de cada carpeta baggingX, o bien dentro de la carpeta principal si no hay bagging, se encuentra:

   a) Un directorio siblings-episodes, conteniendo el fichero de configuración de TILDE para el árbol de operadores (*siblings-episodes.s*).

   b) Un directorio *acción*-episodes, para cada acción, conteniendo conteniendo el fichero de configuración de TILDE para árboles de acciones instanciadas (acción-*episodes.s*).

4. Llamada a ROLLER para generar los ficheros de entrenamiento (*.kb*), correspondientes a los ficheros (*.s*) recién generados. Estos son los ficheros generados tras entrenar a partir de los ejemplos iniciales. Para ello, en primer lugar ha de generarse el conjunto de problemas asignados, que se almacenará en un fichero para poder ser consultado posteriormente. Después, es necesario hacer una llamada por cada problema, almacenándose los resultados en el mismo fichero .kb. El conjunto de entrenamiento estará formado por el mismo número de problemas, pero estos variarán:

   a) Si no hay bagging, se entrena con todos los ejemplos (que empezaran por el prefijo indicado) sin repetir.

   b) Si hay bagging, para cada grupo se hará una selección de N ejemplos, siendo N el número de ejemplos de partida. Esta selección se hará de manera aleatoria, de manera que pueda haber ejemplos repetidos para un mismo grupo. Nótese que el número de problemas empleado será el mismo que en el caso sin bagging.

5. Modificación del fichero de configuración de TILDE, para que contemple la ruta de la máquina en la que se está ejecutando. Se podría hacer manualmente al instalar el sistema en una máquina, pero con esto se automatiza el proceso, haciéndolo más sencillo.

6. Llamada a TILDE para generar los árboles a partir de los ficheros de entrenamiento recién creados.

7. Fin de la ejecución.

Con el fin de clarificar el proceso de entrenamiento, la figura 3.4 muestra el pseudocódigo correspondiente al proceso de entrenamiento de manera resumida.

Finalmente, los ficheros de salida que se tendrán son:

1. Ficheros de configuración de los árboles (.s) (ver Apéndice B.2).

2. Ficheros de entrenamiento (.kb) (ver Apéndice B.3).

3. Árboles de decisión (ver Apéndice B.4).

4. Ficheros de problemas asignados a cada grupo (ver Apéndice B.5).

```
if bagging = FALSE then
    problemas-asignados = problemas-iniciales
else if bagging = TRUE then
    numero-problemas = tamaño(problemas-iniciales)
    for grupo to NUMERO-GRUPOS do
        problemas-asignados[grupo] = aleatorio(numero-problemas,
                                               problemas-iniciales)
while grupo to NUMERO-GRUPOS do
    for problema in problemas-asignados[grupo] do
        roller-entrenamiento()
actualizar(fichero-configuracion-TILDE)
while grupo to NUMERO-GRUPOS do
        tilde()
```

Figura 3.4: Pseudo-código de la fase de entrenamiento llevada a cabo por el programa Entrenador.

3.3.2. Script de automatización de la evaluación para varios dominios

Al igual que el anterior, este programa ha sido implementado en Python y está formado por unas 185 líneas de código.

Su finalidad es la de permitir ejecutar todas las pruebas con una sola llamada. La razón de su creación es simplificar la parte de experimentación, ya que era necesario emplear varios algoritmos, aplicados a diferentes dominios y con diferente número de bags, lo que daba a un número muy elevado de llamadas que era necesario realizar. A ello hay que sumar el hecho de no poder predecir cuándo va a terminar una prueba (porque en unos casos se tarda menos que en otros), lo que hacía muy tedioso el hecho de tener que comprobar cuándo terminaba cada una para poder comenzar la siguiente. Todo ello hizo necesario automatizar todo este proceso para mejorar la eficiencia con la que se obtienen los planes.

El script principal *main_script.py* recibe como entradas:

1. Ruta de la carpeta de dominios: la que contiene todos los dominios que se van a probar.
2. Tiempo máximo para cada problema en entrenamiento.
3. Si se quiere hacer entrenamiento: (0=no; 1=sí).
4. Si se quiere hacer validación (test): (0=no; 1=sí).

La razón de permitir separar la ejecución de entrenamiento de la de validación es poder ejecutar de forma individual cada una de las partes. Esto

permite, por ejemplo, hacer varias pruebas de validación con condiciones diferentes, sin necesidad de repetir el proceso de entrenamiento. De esta manera, en una llamada se podría hacer entrenamiento y validación, después modificar las condiciones de test y realizar una segunda llamada de validación con las nuevas condiciones, aplicadas a los árboles ya creados anteriormente.

Dentro del propio código, existen algunas variables de interés que definen aspectos de las pruebas. Se ha optado por ponerlas dentro del código porque para las pruebas realizadas han sido valores fijos, pero es posible cambiarlas si así se requiere. Son:

1. Lista de dominios que se van a evaluar.
2. Nombres de los ficheros de dominio: los ficheros .pddl en ocasiones tienen un nombre que no se corresponde con el de su dominio. Por ejemplo, para el dominio *Parking*, se ha empleado parking-new.pddl.
3. Prefijos de los problemas de entrenamiento.
4. Prefijos de los problemas de validación.
5. Número de bags que se van a usar en cada prueba. Por ejemplo, la lista: ["2","5"] indica que se van a hacer dos pruebas, una con 2 bags y, la otra, con 5.
6. Tiempo máximo por problema para test.
7. Número del algoritmo (o algoritmos) de bagging que se va a usar. Los números de los algoritmos implementados en este proyecto son: 31 (TBAP) y 32 (MQTBP).
8. Variable para indicar si se quiere mostrar una traza de la ejecución.

Hay que mencionar que las listas han de presentar los dominios en el mismo orden. Así, si en una ejecución del programa el dominio *Depots* ocupa la segunda posición en la primera lista, los valores de las demás listas asociados a ese dominio han de ocupar esa posición para que el programa funcione correctamente.

El proceso de ejecución sigue los siguientes pasos:

1. Lectura de parámetros de entrada.
2. Si se ha solicitado entrenamiento: se llama a roller_trainer (programa entrenador) para cada dominio y número de bags. Así, con los dominios *Blocksworld* y *Depots* y bags ["2", "3", "5"] se harán 6 llamadas.
3. Si se ha solicitado validación, para cada dominio:
   a) Se cargan los problemas de test.

b) Se genera el directorio de resultados.

c) Para cada prueba (nº de bags) y para cada algoritmo se llama a ROLLER para cada uno de los problemas indicándole el fichero de salida para los resultados. Si el problema se ha resuelto correctamente, ROLLER se encargará de generar el fichero de salida con los resultados. En caso de que se agote el tiempo, *main_ script* lo anota en el fichero de rechazados.

La salida generada por este programa será:

1. Estructura de directorios, con la misma información que generaba *roller_ trainer*, para cada uno de los dominios y pruebas.

2. Carpeta de resultados por dominio.

3. Resultados para cada dominio, indicando en un fichero separado cada prueba realizada para un dominio, número de bags y algoritmo empleado. Cada uno de estos ficheros (detallados en el apéndice B.6), por tanto, contendrá toda la información referente a esa prueba para todos los problemas de validación correspondientes. En caso de que no se haya resuelto ningún problema, este fichero no aparecerá.

4. Ficheros de problemas rechazados (detallados en el apéndice B.7), indicando qué problemas de los asignados se han rechazado por falta de tiempo. Habrá un fichero por cada prueba, al igual que en el punto anterior. Este fichero siempre ha de existir, de manera que si no se generase el fichero de resultados, se podría comprobar con el fichero de rechazados que ninguno de ellos se ha resuelto.

El pseudocódigo correspondiente se muestra en la figura 3.5.

Script de automatización (R, T, E, V)

R: Ruta de los dominios
T: Tiempo máximo de entrenamiento por problema
E: Flag de Entrenamiento
V: Flag de Validación

```
lista-dominios = ["blocksworld", "depots", ...]
lista-pddl = ["blocksworld", "Depots", ...]
lista-prefijos-ent = ["train-ensemble", "train-ensemble", ...]
lista-prefijos-val = ["ipc7", "pfile", ...]
lista-numero-bags = ["2", "3"]
tiempo-max-test = 900
lista-algoritmos-bagging = ["31", "32"]
depuración = FALSE

if leer-parametros-entrada = ERROR then
    imprimir-error()
    salir()

if E = TRUE then
    for dominio in lista-dominios do
        for bag in lista-numero-bags do
            carpeta-destino = generar-destino (dominio)
            prefijo-entrenamiento = generar-prefijo-ent (dominio,
                                        lista-prefijos-ent)
            entrenador (dominio, carpeta-destino, bag, prefijo-entrenamiento,
                        T, depuración)

if V = TRUE then
    for dominio in lista-dominios do
        lista-problemas = cargar-problemas-validación()
        generar-directorio-resultados()
        for bag in lista-numero-bags do
            if bag = 1 then
                for problema in lista-problemas do
                    /*El número 10 se corresponde al algoritmo sin bagging.*/
                    roller-validación(10, problema)
            else
                for algoritmo-bagging in lista-algoritmos-bagging do
                    for problema in lista-problemas do
                        roller-validación(algoritmo-bagging, problema)
```

Figura 3.5: Pseudo-código del script de automatización de experimentos.

Capítulo 4

Experimentación

Resumen: En este capítulo se mostrarán los resultados de algunas de las pruebas realizadas, indicando las conclusiones más relevantes que puedan ser extraídas de ellas.

4.1. Condiciones de las pruebas

Los resultados de las pruebas que se van a mostrar en la siguiente sección se han generado bajo ciertas características, que se definen a continuación.

1. Dominios:
   - *a*) Blockworld.
   - *b*) Depots.
   - *c*) Parking.
   - *d*) Rovers.
   - *e*) Satellite.
2. Tiempo de entrenamiento máximo por problema:
   - *a*) 120 segundos.
3. Tiempo de validación máximo por problema:
   - *a*) 900 segundos.
4. Algoritmos empleados:
   - *a*) Depth-First H-Context Policy:
     1) Un sólo árbol.

   b) Tree-Bagging Aggregation Policy:
      1) 2 árboles.
      2) 3 árboles.
      3) 5 árboles.
      4) 10 árboles.
      5) 20 árboles.
   c) Multiple Queue Tree Bagging Policy:
      1) 2 árboles.
      2) 3 árboles.
      3) 5 árboles.

5. Aspectos evaluados, reflejados en las tablas:
   a) Problema / Coste.
   b) Problema / Relación de tiempo de matching entre tiempo total. *El tiempo de unificación o 'matching" se define como, dado un estado del árbol de búsqueda y su contexto, el tiempo de obtención de la ordenación de acciones utilizando los árboles correspondientes.*
   c) Problema / Tiempo de CPU hasta la resolución del problema.
   d) Límite de coste / Porcentaje de problemas resueltos.
   e) Límite de tiempo / Porcentaje de problemas resueltos.

La nomenclatura empleada en las gráficas para hacer referencia a cada caso probado sigue la estructura "*número_de_bags-algoritmo*".

Todas las pruebas se han realizado en una sola máquina, con 4 núcleos de 3GHz, 2048 KB de memoria caché y 3 GB de memoria principal.

4.2. Dominios de evaluación

Se ha creído conveniente incluir esta sección para aportar algo de información sobre en qué consiste cada uno de los dominios que se han evaluado, de manera que cualquier lector pueda, al menos, entender la complejidad de cada uno de ellos y las diferencias que presentan entre sí.

No se pretende con esta sección entrar muy en detalle en cada uno de los dominios. Éstos, por otro lado, son dominios "clásicos" en planificación automática, por lo que no será difícil encontrar información adicional fuera de este documento, por aquellas personas que puedan estar interesadas en entrar en profundidad.

Una descripción más detallada sobre los tipos, predicados y acciones de cada dominio, puede consultarse el Apéndice C.

4.2.1. Blocksworld

El dominio Blocksworld, conocido también en español como "Mundo de los bloques", representa un espacio en el que hay varios bloques ordenados de una manera determinada y hay que conseguir ordenarlos de otra forma, usando para ello un brazo articulado encargado de cambiar su disposición.

La figura 4.1 muestra un ejemplo de los estados inicial y final en el mundo de los bloques.

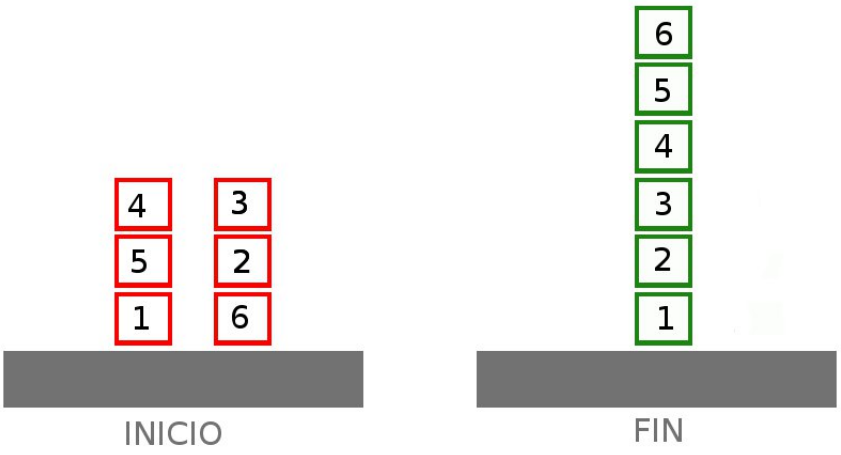

Figura 4.1: Ejemplo de representación del mundo de los bloques.

El orden de cada bloque viene determinado por su posición en el espacio, indicando así si está directamente sobre el suelo, si tiene algún otro bloque encima o está libre, etcétera.

El brazo, por su parte, sólo puede transportar un bloque cada vez. Además, sólo podrá coger los bloques que no tengan ningún otro encima y sólo podrá dejarlo encima de otro bloque que tampoco tenga ningún otro encima o bien en el suelo.

El mundo de los bloques supone una simplificación de ciertos problemas que se pueden encontrar en la vida real. La figura 4.2 muestra un ejemplo de puerto comercial, en el que los bloques son contenedores de mercancías, que han de ser apilados unos encima de otros de manera ordenada en base a ciertos criterios, de forma que se agilice su salida del puerto (los contenedores que tengan que salir antes se colocarán preferentemente arriba), pero permitiendo al mismo tiempo la entrada de nuevos contenedores al puerto (optimizando el espacio y el uso de las grúas).

Figura 4.2: Ejemplo real del "Mundo de los bloques": puerto comercial.

La importancia de la planificación automática en este dominio radica en la posibilidad de obtener planes que consigan pasar de la ordenación inicial de los bloques a la ordenación deseada, usando el brazo el menor número de veces posible, lo que supondrá un ahorro de tiempo y de coste. Supongamos un caso real (por ejemplo el puerto comercial ya comentado) en el que el brazo articulado lleva asociado un coste energético inherente para manipular un bloque: resulta evidente que, cuanto menos se tenga que usar el brazo, habrá menor gasto de energía, menor desgaste (más tiempo de vida) para sus piezas, menor coste de mantenimiento, menor riesgo de accidentes y, por supuesto, menor tiempo empleado. Con el uso de la planificación automática se pretende así encontrar un plan hasta la solución en que se reduzca el número de desplazamientos innecesarios del brazo.

4.2.2. Depots

El dominio "Depots" (en español "Almacenes") hace alusión a un entorno de almacén de mercancías. A él llegan camiones cargados de cajas, que han de ser descargadas y colocadas en palés.

La idea fundamental se basa en ser capaces de transportar mercancías entre varios almacenes, optimizando la carga y descarga de cajas de los camiones cuando éstos están disponibles.

Con la aplicación de la planificación automática, se busca optimizar estos procesos, decidiendo previamente cuándo ha de realizarse la carga/descarga

de una caja de un almacén, de manera que se mejore la eficiencia de todo el sistema.

Se puede entender el dominio Depots como una ampliación del mundo de los bloques, centrada en la logística.

4.2.3. Parking

Este dominio representa un aparcamiento de coches, en el que cada posición viene indicada por un "bordillo", en frente del cual puede haber a la vez dos coches, uno bien aparcado y, el otro, en doble fila.

La finalidad es conseguir que, partiendo de una disposición inicial de vehículos en el aparcamiento, se termine con uno o más coches situados en otro lugar. De forma práctica, la utilidad de resolver este dominio con planificación automática se basa en lograr sacar un coche aparcado, que puede estar taponado por otro en doble fila, empleando para ello el menor número de movimientos del resto de coches, ahorrando combustible y, sobre todo, reduciendo el tiempo derivado de los desplazamientos.

Puesto que cada coche puede estar en dos tipos de posiciones, bordillo o doble fila, las acciones de este dominio van encaminadas a generar un desplazamiento en los coches.

4.2.4. Rovers

Un Rover es un vehículo de exploración espacial, encargado de llevar a cabo misiones de reconocimiento en lugares aislados, bien sea ayudando al transporte de la tripulación (como por ejemplo los vehículos lunares) o tomando muestras de forma autónoma (como los rovers de Marte).

Figura 4.3: MER (*Mars Exploration Rover* - *Rover de Exploración de Marte*).

Son estos últimos los que atañen a la planificación automática, que será la encargada de optimizar los comportamientos del robot, de manera que

haga un uso eficiente de los recursos de que dispone durante sus misiones.

Estas misiones incluyen labores tales como la recogida de muestras del terreno o la toma de fotografías. Para ello, el vehículo ha de ser capaz de desplazarse, cubriendo la mayor parte del espacio a su alcance, decidiendo en consecuencia adónde moverse, de manera que utilice eficientemente el combustible del que dispone y asegurándose de que la información que recaba es relevante para la misión que le ha sido encomendada. Es ahí donde la planificación automática puede contribuir a planificar las acciones del robot intentando mejorar su rendimiento.

Para poder hacer todo eso, un rover ha de disponer, además de la mecánica necesaria para desplazarse por entornos escabrosos, de diversos sensores, cámaras y dispositivos de comunicación.

4.2.5. Satellite

Este dominio surgió como una manera de hacer frente a la programación del tiempo que permitiera a los satélites empleados por la agencia espacial de Estados Unidos (NASA (National Aeronautics and Space Administration)) realizar de forma eficiente observaciones espaciales.

El funcionamiento de los satélites se podía resumir en que, mediante una serie de instrumentos de los que disponían, se encargaban de realizar observaciones del espacio, apuntando a diferentes lugares. Los datos generados eran almacenados hasta que surgía una oportunidad de comunicación con la estación de la Tierra.

Con la aplicación de la planificación automática se buscaba mejorar el rendimiento de los satélites, optimizando el tiempo en el que debían realizarse las comunicaciones; de manera que, mientras tanto, los satélites pudieran tomar la mayor cantidad de datos posible que les permitiera su capacidad interna de almacenamiento.

Existen diferentes formalizaciones del dominio. Algunas, más completas que otras, contemplan por ejemplo la capacidad de combustible del satélite, la cantidad de datos que puede almacenar como máximo en un momento dado e incluso el tiempo que se tarda en tomar un tipo de muestra.

4.3. Resultados de las pruebas

En esta sección se mostrarán los resultados de las pruebas realizadas para cada uno de los dominios.

4.3.1. Blocksworld

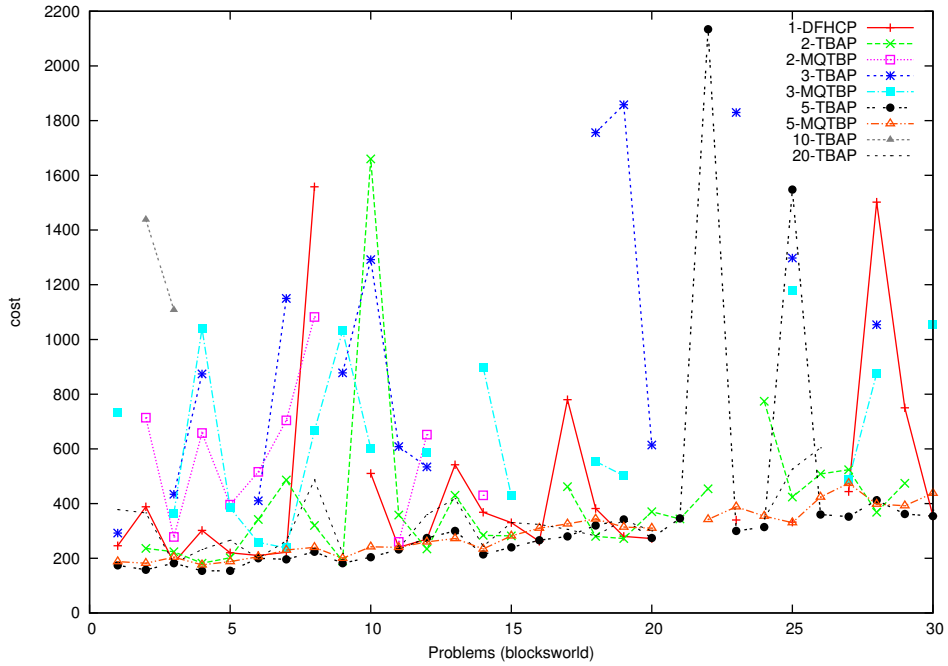

Figura 4.4: Problema / Coste - Blocksworld.

La figura 4.4 muestra cómo, para el dominio Blocksworld, el coste de los problemas resueltos varía de un algoritmo a otro. En ella se puede ver, además, como algunos algoritmos no consiguen resolver todos los problemas.

Por ejemplo, usando 2 bags y el algoritmo de múltiples listas (MQTBP) sólo se resuelven 10 problemas de 30 y, además, con un coste elevado. Esto puede ser debido a que al asignar los problemas de entrenamiento a cada bag de manera aleatoria, con sólo 2 bags no se tengan grupos significativos de entrenamiento que generen un buen clasificador. Por otro lado, si se usa el algoritmo TBAP, también con 2 bags, se consigue mejorar el número de problemas resueltos, ya que ahora se están combinando las pautas dadas por los árboles de cada una de las bags, lo cual aumenta también la calidad del clasificador.

Con 3 bags, sin embargo, ocurre lo contrario. Con el algoritmo de agregación se resuelven menos problemas (3 menos) que con el de múltiples listas, y los que se resuelven, tienen un coste mucho mayor.

Para 5 bags, los resultados son muy similares entre un algoritmo y otro, siendo menores los costes para el algoritmo de agregación (TBAP). Además, ambos tienen una tasa de problemas resueltos muy alta, resolviendo 30 problemas (todos) el algoritmo TBAP, y 29 problemas el algoritmo MQTBP.

A partir de 10 bags, sólo se han mostrado los casos del algoritmo TBAP.

La razón es que el algoritmo MQTBP no ha sido capaz de dar solución a ninguno de ellos en el tiempo máximo asignado por problema. En este sentido, se ve una clara ventaja en el uso del primer algoritmo, ya que además de ser más rápido ha demostrado dar solución a una gran cantidad de problemas en el caso de 20 bags.

NOTA: Aunque esto se verá próximamente, cabe adelantar que ha sido una tendencia para todos los dominios el hecho de que el segundo algoritmo (MQTBP) no haya sido capaz de resolver ningún problema para casos de 10 bags en adelante, por lo que no se hará nuevamente referencia a este hecho.

Por último, al comparar con el uso de ROLLER sin bagging (1-DFHCP), vemos cómo con 5 bags se resuelven bastantes problemas más, y con menor coste (en algunos casos bastante significativo).

De esta figura se puede concluir que para las pruebas realizadas, se obtienen mejores resultados con 5 bags que sin bagging, pero para un menor número de bags los costes tienden a ser más elevados.

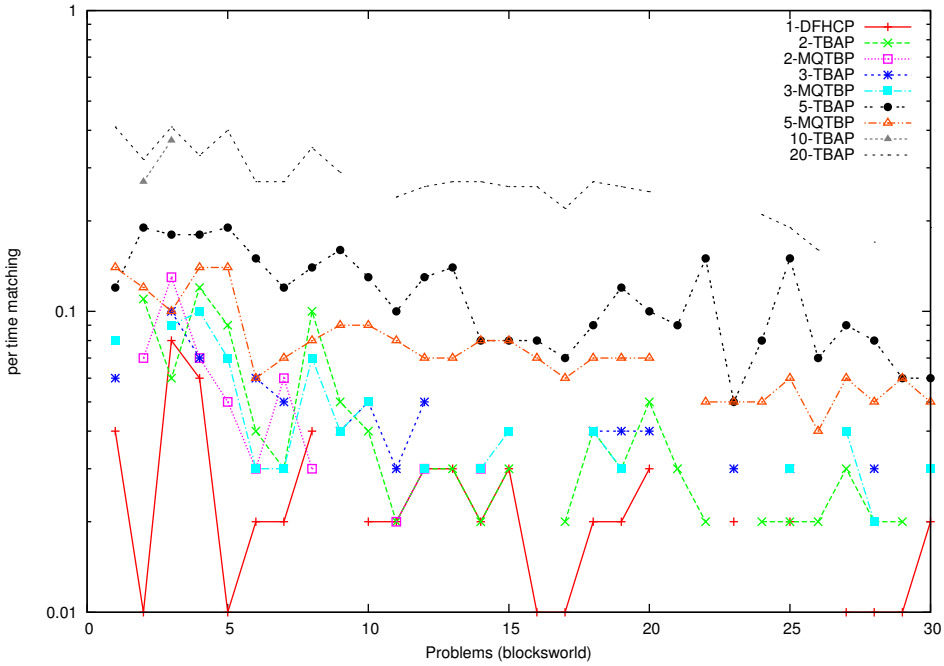

Figura 4.5: Problema / Relación de tiempo de matching entre tiempo total (versión logarítmica) - Blocksworld.

La gráfica 4.5 muestra cómo el tiempo de matching entre el tiempo total, representa un menor valor para el uso de ROLLER sin bagging. Esto es lo esperado, ya que sólo se tiene un árbol de decisión del que extraer información.

Por otro lado, muestra cómo los ratios más elevados se corresponden a

las pruebas realizadas para 10 y 20 bags. De nuevo, es lo normal, ya que se cuenta con un mayor número de árboles de decisión.

Respecto a la comparación entre el algoritmo TBAP y el MQTBP, para un mismo número de bags, se ve cómo la tendencia es que sea mayor el ratio para el algoritmo de agregación (TBAP). Esto se debe a que hay que hacer matching para cada uno de los árboles, para posteriormente poder combinar la información.

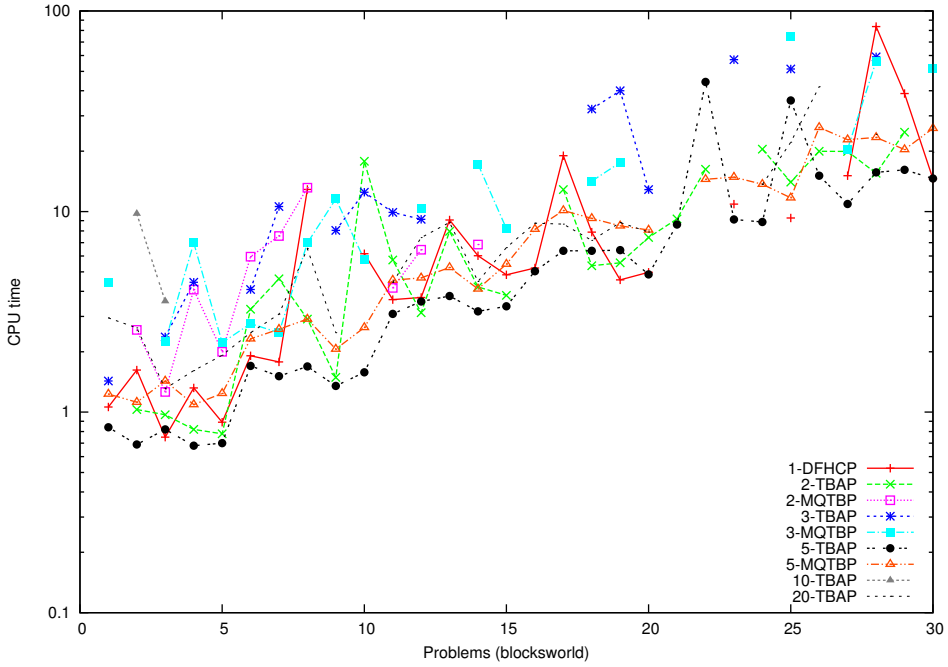

Figura 4.6: Problema / Tiempo de CPU - Blocksworld.

La figura 4.6 muestra cómo los menores tiempos por problema se obtienen para el algoritmo TBAP con 5 bags. En las tablas de "Problema/Relación de tiempo de matching entre tiempo total" se vio cómo con 5 bags los ratios eran más elevados para 5 bags que para un número menor de agrupaciones. Sin embargo, en la figura 4.6 se ve cómo esto ha merecido la pena, ya que aunque la mayor parte del tiempo sea dedicado a hacer matching empleando los árboles de decisión, una vez realizada esta tarea el tiempo de resolución del problema decrece enormemente.

Con 5 bags y el algoritmo MQTBP la tendencia también es positiva, aunque los tiempos sean algo mayores.

Con 20 bags, se observa cómo los tiempos son intermedios, no mejorando el caso anterior, pero tampoco siendo de los peores. Esto indica que, a pesar de contar con más árboles de decisión, el tiempo total no se ve afectado de forma directa por este hecho.

Para el resto de casos, no hay una tendencia clara, variando mucho de un problema a otro. Esto indica que los clasificadores generados para menos de 5 bags no son demasiado buenos, y por ello, mientras que funcionan de forma eficiente en algunos casos (por ejemplo para el problema 25 se mejoran los tiempos de 5-TBAP con 2-TBAP), en otros muchos consiguen resolver los problemas en un tiempo demasiado elevado en comparación con el empleado por los casos que emplean 5 bags.

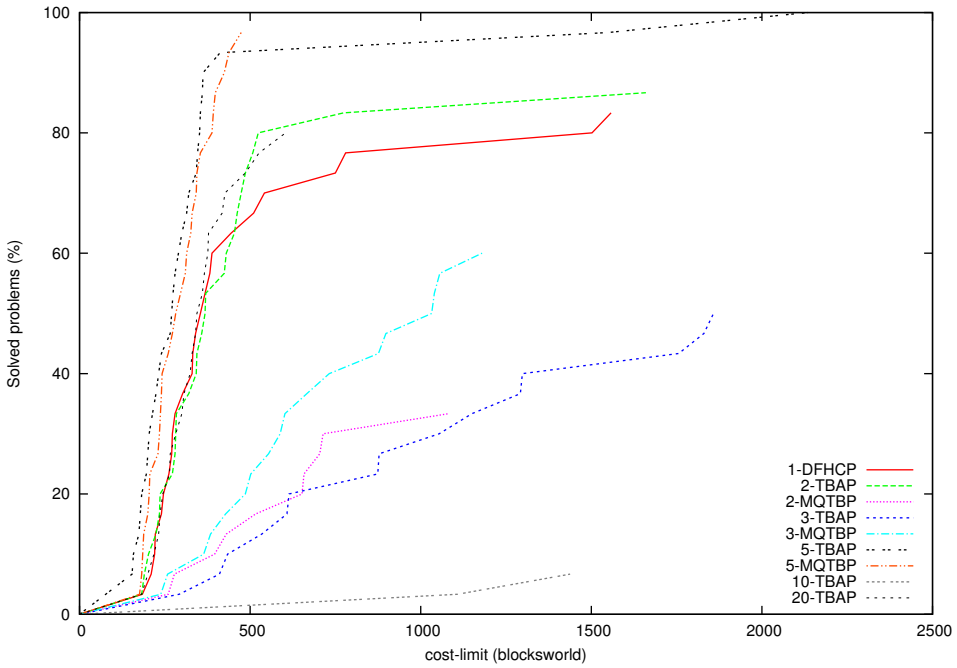

Figura 4.7: Límite de coste / Porcentaje de problemas resueltos - Blocksworld.

La figura 4.7 muestra la relación "Límite de coste / Porcentaje de problemas resueltos". Con este tipo de gráfica se pretende dar a conocer las diferencias entre unos algoritmos y otros en cuanto a la cantidad de problemas que han sido capaces de resolver y el coste máximo de los planes generados. Para visualizar correctamente la representación, hay que comparar los valores de los diferentes algoritmos (eje vertical) para un mismo valor de coste (eje horizontal). De esta forma, se podrá afirmar que un algoritmo ha funcionado mejor que otro para problemas de coste igual o menor que un valor, cuando la línea que representa a ese algoritmo se encuentre más arriba que las del resto de algoritmos, para dicho valor de coste.

La figura 4.7 muestra de forma bastante clara las ideas que se han ido comentando para las demás tablas de este dominio. Se ve cómo los algoritmos que emplean 5 bags resuelven un porcentaje muy alto de problemas (cercano al 100 %) con un coste inferior a 500. Es especialmente notable el

caso 5-MQTBP, a pesar de que no alcance el 100 % de problemas resueltos. Con 5-TBAP la tendencia es la misma, y en este caso sí resuelve todos los problemas, pero generando en ocasiones planes de mucho amyor coste.

Las pruebas realizadas con 2-TBAP para este dominio han resultado ser bastante positivas, resolviendo en torno al 80 % de los problemas con un coste en torno a 500. Una tendencia similar es presentada por el caso 20-TBAP.

El caso de ROLLER sin bagging presenta una tendencia similar a la de los casos mencionados, pero sólo para planes de bajo coste. Cuando el coste de los problemas es mayor, el porcentaje de problemas resueltos es bastante inferior al presentado por 2-TBAP o por 20-TBAP.

El resto de casos muestran claramente que no son rentables, ya que resuelven porcentajes de problemas muy bajos (inferiores al 50 %) y con costes muy elevados. Es significativa en este caso la baja eficiencia del caso 10-TBAP, que contrasta con los buenos resultados mostrados por ese algoritmo para un número diferente de bags.

4.3.2. Depots

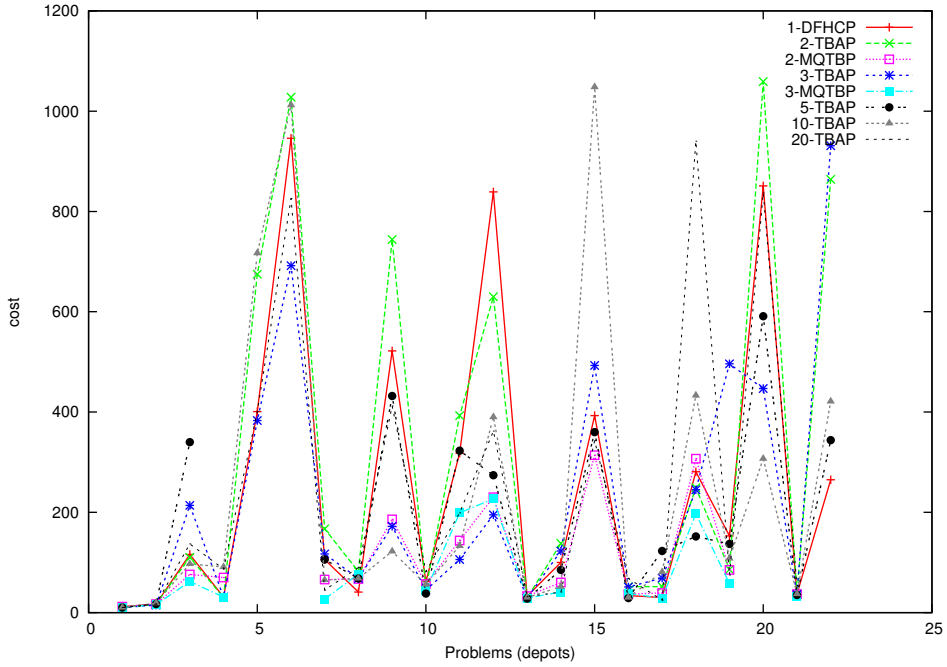

Figura 4.8: Problema / Coste - Depots.

La gráfica 4.8 muestra cómo los problemas de este dominio son resueltos por los distintos algoritmos de forma bastante precisa, puesto que, aunque con unos se obtienen menores costes que con otros, se observa claramente cómo con todos ellos la gráfica tiende a mostrar los mismos picos para cada

problema.

Esto indica que, salvo algún caso aislado, los clasificadores generados tienen suficiente información como para resolver cada problema de la mejor manera posible. Esto difiere de lo que ocurría con el dominio anterior, en el cual algunos algoritmos resolvían determinados problemas con costes demasiado elevados respecto a las soluciones propuestas por otros algoritmos, siendo muestra de que esos clasificadores se habían entrenado muy bien para resolver unos problemas, pero no contaban con la información suficiente como para resolver otros de forma eficiente.

En cuanto a la comparativa entre unos casos y otros, no se observa una tendencia clara de un sólo algoritmo que mejore los costes de todos los demás. Así, en unos casos, hay determinados problemas que sólo se resuelven con unos algoritmos y con otros no, o casos de problemas que son resueltos muy bien por un algoritmo, que sin embargo es el peor para resolver el siguiente problema.

Esto último puede ser una muestra de que los conjuntos de entrenamiento pueden estar afectando negativamente a la generación de los árboles, que están predispuestos a la resolución de determinados problemas por encima de otros. Es debido a la generación aleatoria de los conjuntos de entrenamiento para cada bag, a diferencia de lo que ocurre cuando se usa ROLLER sin bagging, en cuyo caso se entrena con todos los problemas disponibles, sin repetir.

Otro dato significativo, es que no se ha conseguido resolver ningún problema con 5 bags y múltiples listas simultáneas, ya que el tiempo máximo por problema se agotaba antes de generar la solución, por lo que dicho problema era descartado. Para el dominio anterior se ha visto cómo la tendencia a mejorar las soluciones aumentaba de forma directa al número de bags empleado, por lo que hubiera sido interesante haber obtenido resultados para un mayor número de conjuntos. Sin embargo, no ha sido posible, partiendo de la premisa de detener la generación de soluciones que tardasen más de 900 segundos.

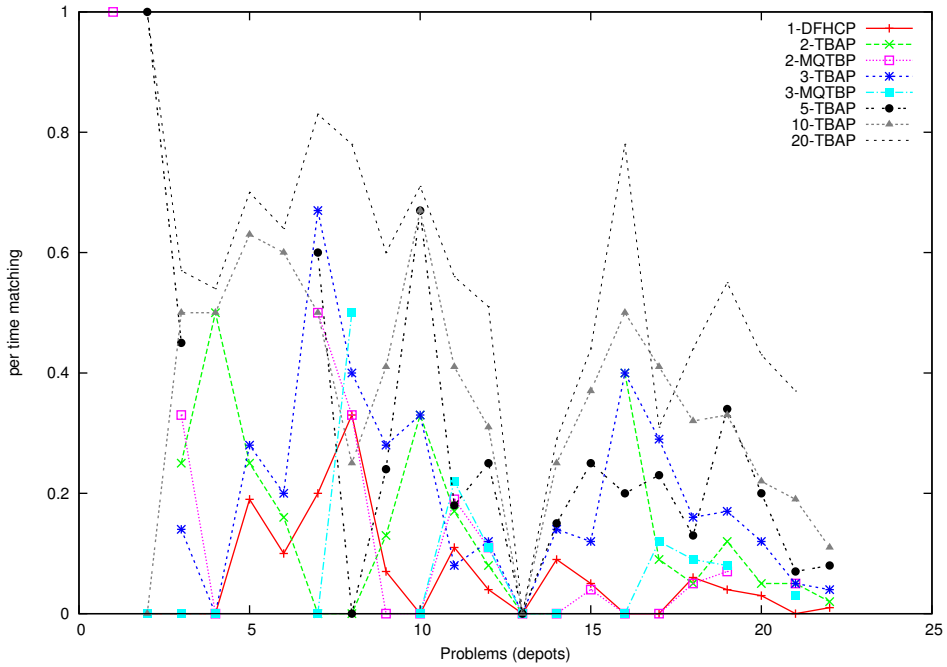

Figura 4.9: Problema / Relación de tiempo de matching entre tiempo total - Depots.

Como se puede ver en la figura 4.9, de la misma forma que ocurría con el dominio anterior, los casos que han empleado el algoritmo TBAP muestran un mayor tiempo de matching respecto a los que han usado el algoritmo MQTBP, siendo mayores estos tiempos para los experimentos realizados con un mayor número de bags.

Igualmente, el uso de ROLLER sin bagging, presenta los menores tiempos de matching, ya que emplea un único árbol de decisión.

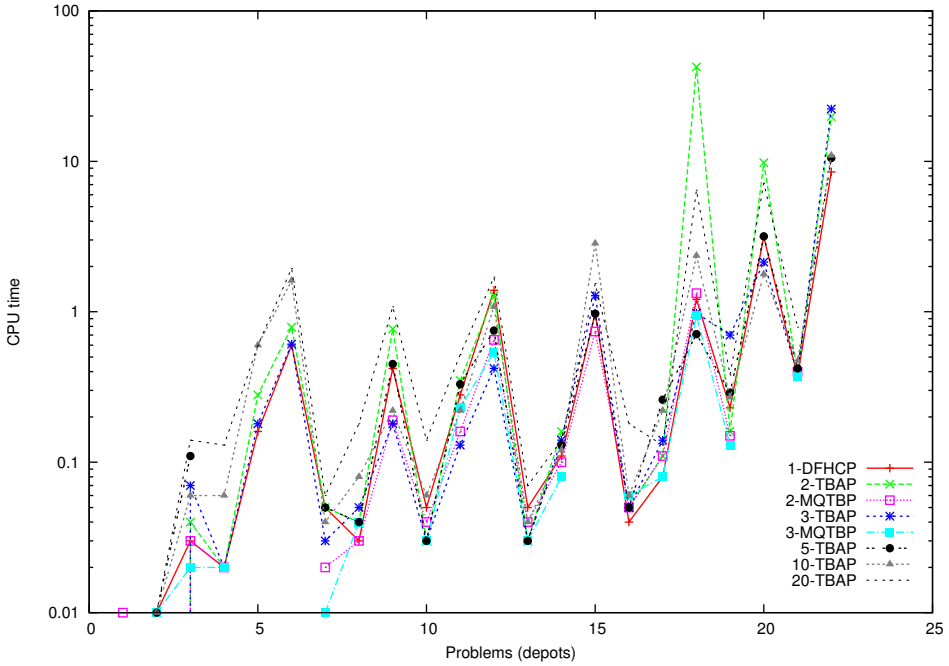

Figura 4.10: Problema / Tiempo de CPU (versión logarítmica) - Depots.

La gráfica 4.10 está relacionada con la primera (Problema/Coste). Se puede ver una silueta casi idéntica, indicando que para los problemas cuya solución presenta mayor coste, el tiempo necesario para obtenerla ha sido también mayor, y aquellos problemas con una solución más sencilla (menor coste), han necesitado menor tiempo.

Esto ya ocurría también para el dominio anterior. Es lo normal, ya que un plan con muchos operadores habrá necesitado mayor tiempo de cómputo para tratar de obtener los mejores.

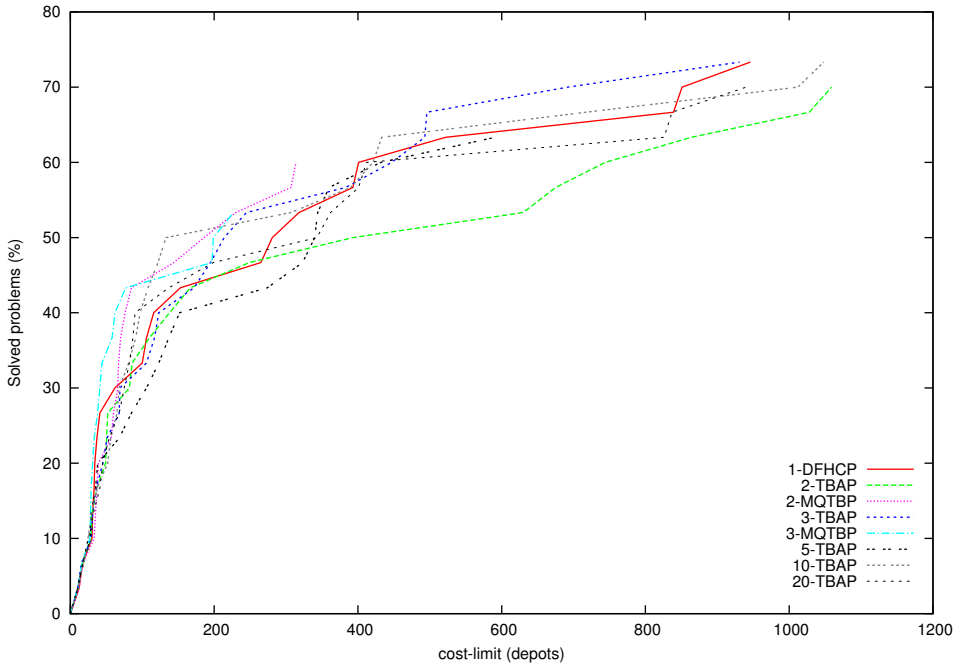

Figura 4.11: Límite de coste / Porcentaje de problemas resueltos - Depots.

La gráfica 4.11 muestra cómo, en este dominio, a diferencia de lo que ocurría con Blocksworld, determinados problemas son resueltos mediante planes con un coste elevado. Esto puede indicar que este dominio es algo más complicado que el anterior, ya que todos los casos evaluados presentan una tendencia similar.

El mejor caso es el 3-TBAP, seguido por el 1-DFHCP, que consigue resolver el mayor porcentaje de problemas de todas las pruebas para este dominio, entre el 70 % y el 80 %. Esto nos muestra que, al menos para este dominio, el uso de ROLLER tradicional puede ser positivo si se quiere conseguir una tasa elevada de problemas resueltos.

Después, hay dos casos significativos. Por un lado, el 20-TBAP, que para un mismo coste que los casos ya mencionados, consigue un porcentaje de problemas resueltos cercano al 70 %. El otro caso significativo es el 10-TBAP, que consigue porcentajes de éxito similares a los de los dos primeros casos mencionados, eso sí, generando planes con un coste significativamente mayor.

El resto de casos no han sido muy positivos, ya que consiguen porcentajes máximos de problemas resueltos entre el 50 % y el 65 %.

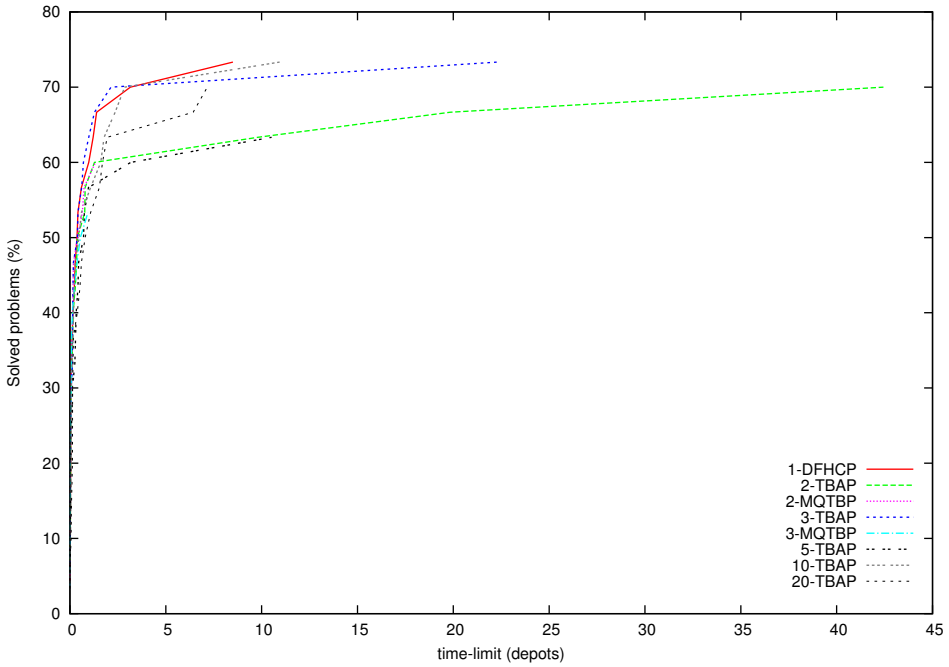

Figura 4.12: Límite de tiempo / Porcentaje de problemas resueltos - Depots.

La figura 4.12 muestra la relación "Límite de tiempo / Porcentaje de problemas resueltos". Con este tipo de gráfica se pretende dar a conocer las diferencias entre unos algoritmos y otros en cuanto a la cantidad de problemas que han sido capaces de resolver y el tiempo máximo empleado para ello. De esta manera, se puede ver, por ejemplo, si para un mismo tiempo un algoritmo ha sido capaz de resolver la mayoría de problemas frente a otro que no, o cuánto ha de incrementarse el tiempo máximo para que un algoritmo resuelva un porcentaje elevado de problemas.

La gráfica 4.12 es bastante interesante. En ella se puede ver cómo para la mayoría de casos probados, con tiempos bajos se consigue alcanzar el máximo porcentaje de problemas resueltos y apenas mejora dedicando más tiempo.

Es decir, a la vista de los datos, si un problema no se ha resuelto en menos de 3 segundos, sería conveniente cortar la ejecución, puesto que es probable que no se consiga resolver. De esta manera, para este dominio se podría ahorrar tiempo de cómputo y poder tratar de resolver una mayor cantidad de problemas en el mismo tiempo, sin necesidad de agotar los 900 segundos por problema de los que se disponía inicialmente.

4.3.3. Parking

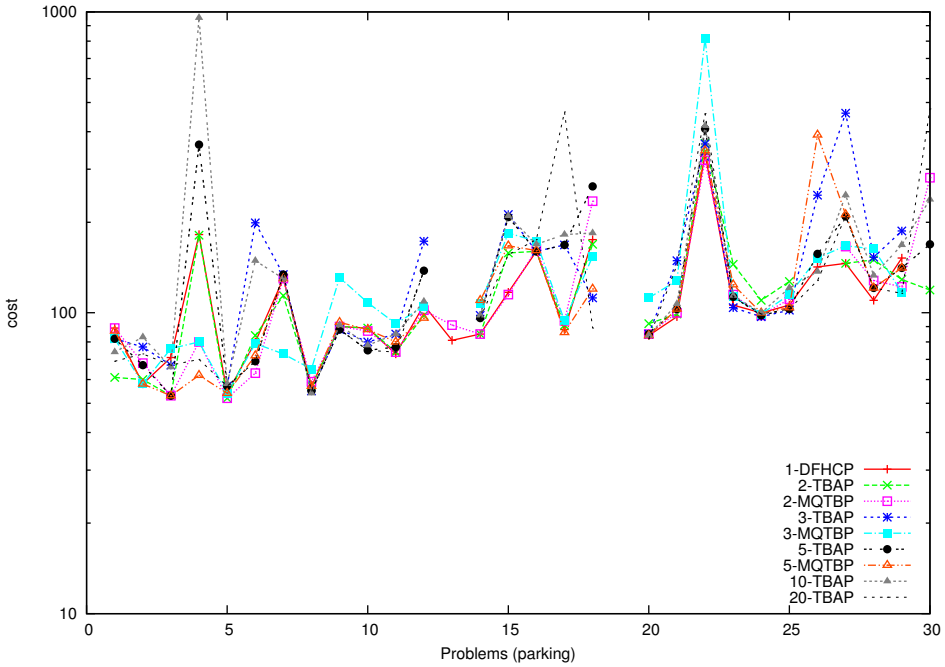

Figura 4.13: Problema / Coste - Parking.

La gráfica 4.13 muestra cómo para algunos problemas, la mayoría de los casos evaluados tienen una solución con un coste muy parecido.

Para otros problemas, sin embargo, hay grandes variaciones entre unos casos y otros. Así, por ejemplo, para el problema 4, con 5-MQTBP se obtiene el menor coste, muy alejado del propuesto por 10-TBAP, o por 5-TBAP.

En general, el caso que muestra menores costes para una gran parte de problemas es el 5-MQTBP. Por contra, el 3-MQTBP muestra costes más elevados que los propuestos por el resto de casos para la mayoría de problemas.

Y al igual que ocurría con el dominio anterior, el uso de ROLLER sin bagging presenta también buenos resultados de forma general, a pesar de que en algunos casos se vea mejorado por otros algoritmos.

Cabe destacar que ahora, a diferencia de lo que ocurría con Depots, sí se han conseguido generar resultados para el caso 5-MQTBP, que antes no resolvía ningún problema en el tiempo máximo asignado. Y no sólo eso, sino que como ya se ha comentado, ha sido de los que mejores resultados ha generado. Esto hace pensar una vez más cómo hubieran sido los planes generados si la máquina empleada hubiera sido capaz de resolver los problemas propuestos en el tiempo disponible para un mayor número de bags con ese algoritmo.

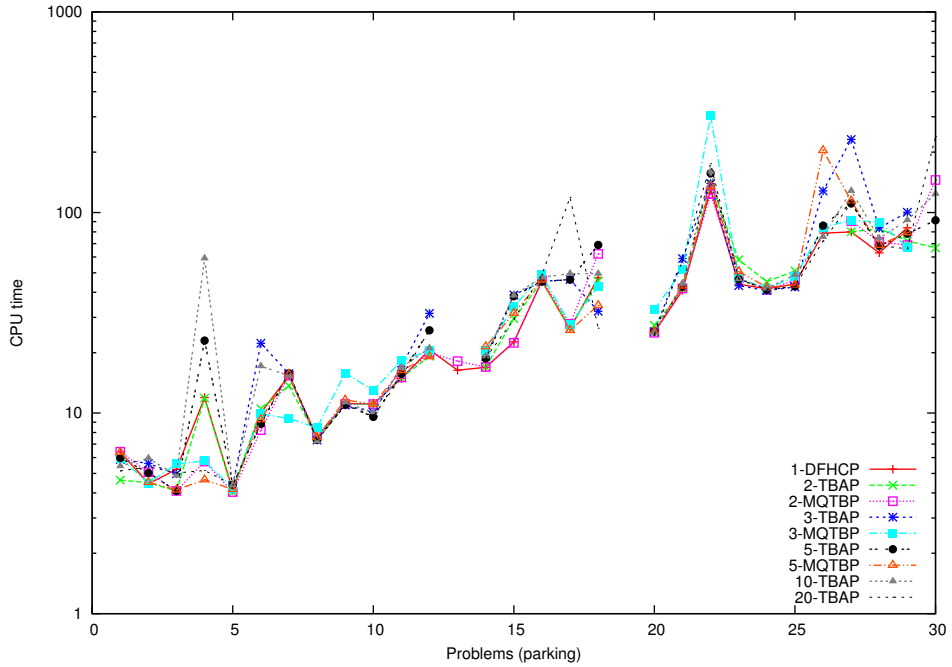

Figura 4.14: Problema / Tiempo de CPU (versión logarítmica)- Parking.

Como ya se ha comentado de manera análoga para los dominios anteriores, la gráfica 4.14 está relacionada con la primera, por lo que no es necesario repetir las mismas conclusiones.

Como dato destacable, está el hecho de ver que para este dominio los últimos problemas requieren un mayor tiempo para ser resueltos, de lo que se puede deducir que su complejidad es también mayor.

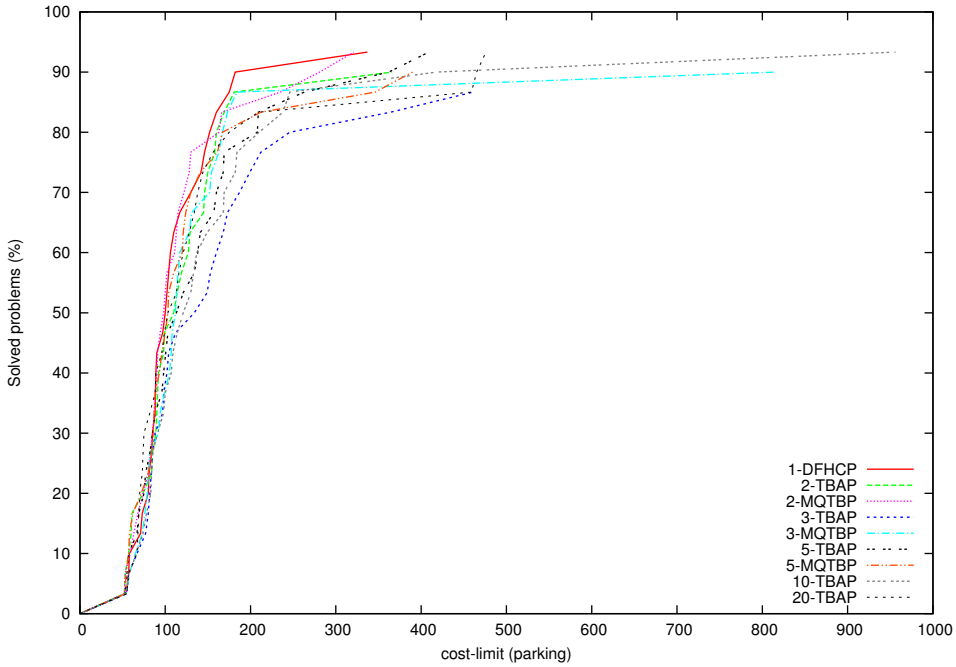

Figura 4.15: Límite de coste / Porcentaje de problemas resueltos - Parking.

La gráfica 4.15 muestra cómo para un coste menor que, aproximadamente, 50, el porcentaje de problemas resueltos es prácticamente idéntico para todos los casos probados. Esto puede indicar que dentro del conjunto de evaluación, hay algún problema muy sencillo que todos los algoritmos resuelven fácilmente.

A partir de ahí, para costes mayores, se comienza a apreciar las diferencias entre unos casos y otros. Así, por ejemplo, para valores entre 100 y 200, 1-DFHCP, 2-TBAP y 2-MQTBP resuelven entre el 80 % y el 90 % de los problemas planteados, mientras que 3-TBAP apenas supera el 50 %.

En valores entre 100 y 200, de nuevo el crecimiento pasa a ser lento, teniendo que ampliar mucho el coste máximo permitido para obtener mejores porcentajes de problemas resueltos.

En comparación con el dominio anterior, se ve cómo ahora para aproximadamente los mismos valores del coste máximo, se obtienen porcentajes de problemas resueltos mucho mayores que antes, lo que puede indicar que el conjunto de problemas de Parking es más sencillo.

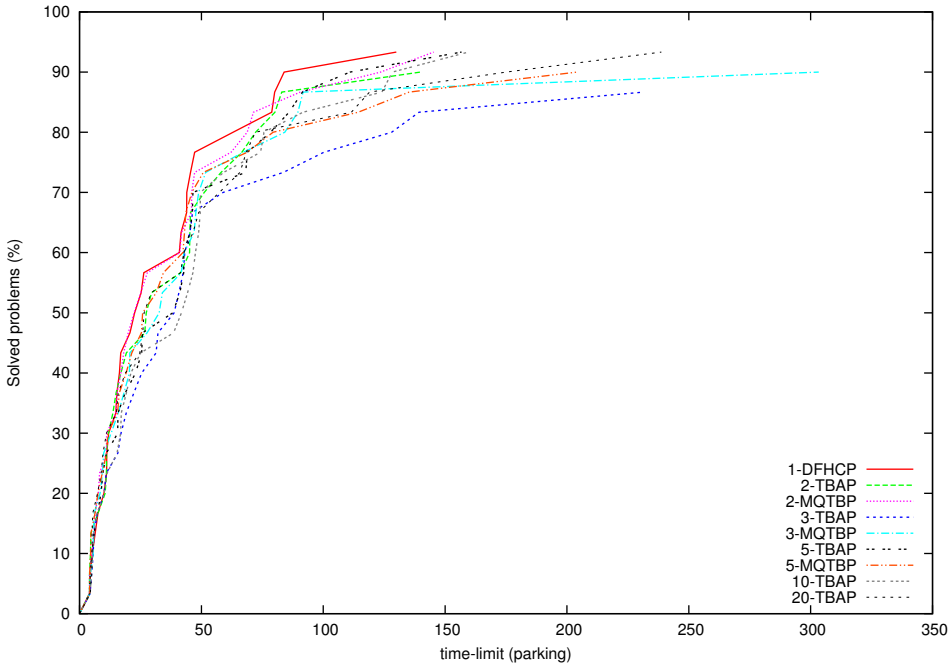

Figura 4.16: Límite de tiempo / Porcentaje de problemas resueltos - Parking.

La gráfica 4.16 muestra cómo ahora, para este dominio, es necesario dedicar mucho más tiempo del que se dedicaba para Depots. Sin embargo, como ya se ha comentado, en Parking estamos obteniendo mayores porcentajes de problemas resueltos, por lo que es preferible este caso ya que, aunque se tarde más, se está asegurando una tasa de éxito mayor.

Así, por ejemplo, a partir de unos 50 segundos, se resuelve en torno a un 70-80 % de los problemas propuestos para la mayoría de los casos.

A partir de ahí, el crecimiento de los valores de la gráfica pasa a ser más lento, requiriendo una mayor inversión en tiempo para mejorar el porcentaje de problemas resueltos.

Los casos que mejores procentajes de problemas resueltos han generado son el 1-DFHCP, el 2-MQTBP, el 5-TBAP y el 10-TBAP, con una diferencia de unos 25 segundos entre el peor y el mejor caso.

4.3.4. Rovers

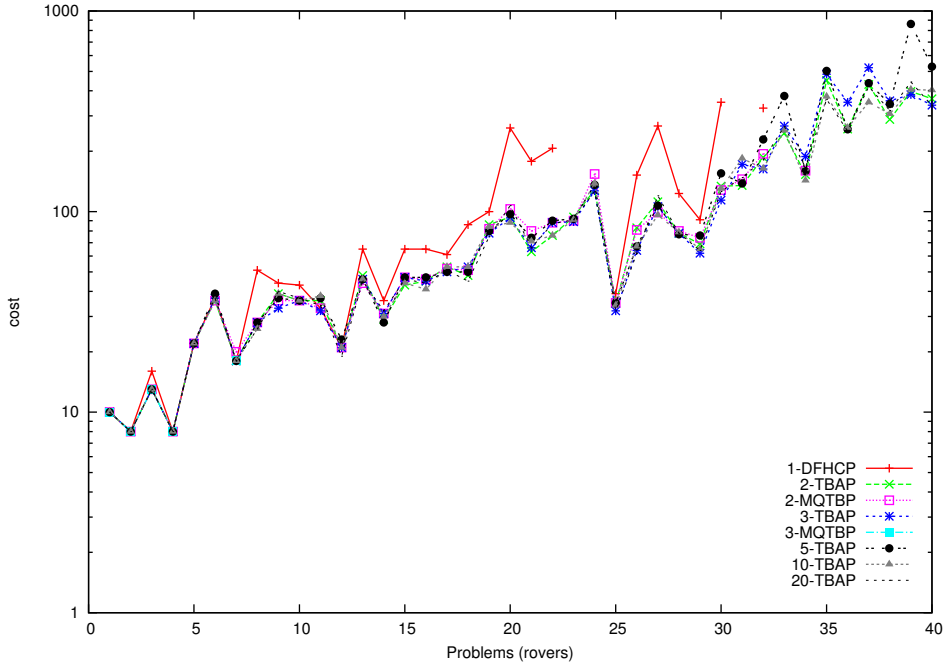

Figura 4.17: Problema / Coste (versión logarítmica) - Rovers.

La gráfica 4.17 muestra cómo para los casos que han usado bagging los costes de los planes generados son prácticamente idénticos, con alguna pequeña diferencia para alguno de los problemas.

Lo más destacable es que, en este caso, los resultados de ROLLER sin bagging son peores para la mayoría de los casos, o iguales, pero nunca mejores. Además, resuelve menos problemas, especialmente los últimos.

En cuanto a los mejores casos, aunque casi no hay diferencia con los demás, son el 3-TBAP y el 10-TBAP.

Con el caso 3-MQTBP se resuelven muy pocos problemas y, al igual que ocurría con Depots, no se consigue resolver ningún problema con 5-MQTBP en el tiempo máximo disponible.

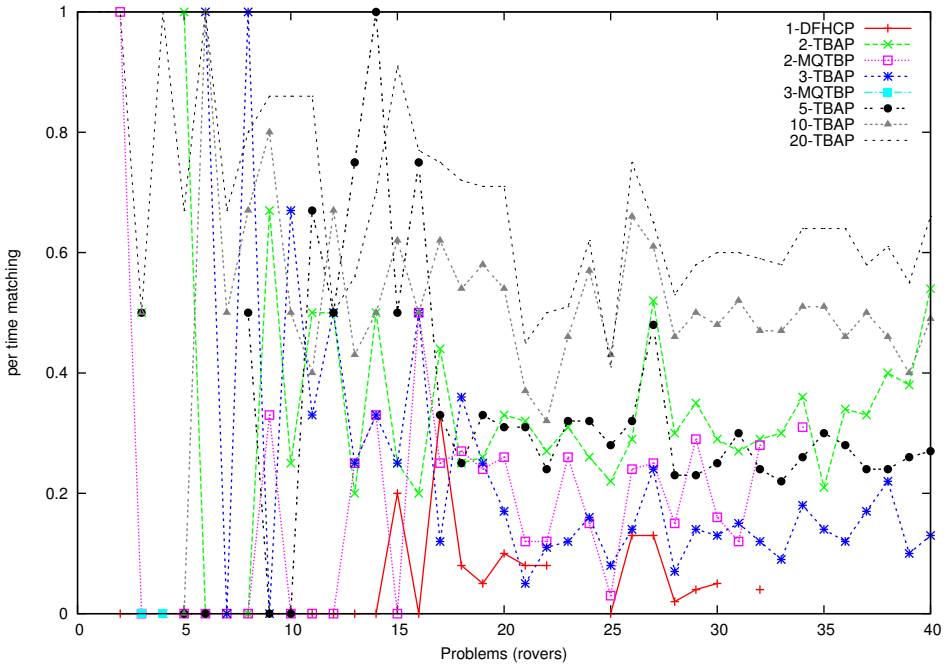

Figura 4.18: Problema / Relación de tiempo de matching entre tiempo total (versión logarítmica) - Rovers.

La gráfica 4.18 muestra la misma tendencia que se venía mostrando para los dominios anteriores. Y es que, de forma general, el ratio de tiempo de matching respecto del total es más elevado para aquellos casos que emplean un mayor número de árboles de decisión. De manea opuesta, el caso de ROLLER sin bagging tiene los menores ratios.

Por otro lado, y en contraposición a la tendencia aparentemente percibida en lo comentado en el párrafo anterior, se puede observar que los dos casos realizados con 2 bags, presentan ratios elevados, superiores a los casos de 3 bags. Especialmente en el caso 2-TBAP. Esto puede ser porque con únicamente dos bags, los árboles generados no sean lo suficientemente buenos o su tamaño sea demasiado grande, provocando la necesidad de emplear un mayor tiempo en el proceso de unificación.

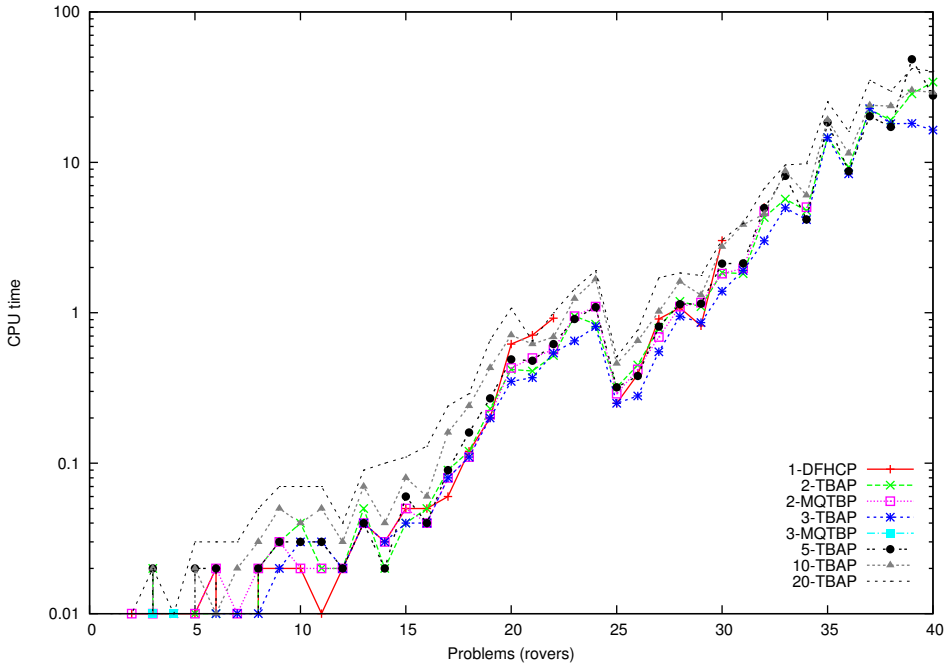

Figura 4.19: Problema / Tiempo de CPU (versión logarítmica) - Rovers.

En la gráfica 4.19, correspondiente a los tiempos dedicados a cada problema, se puede comprobar el hecho de que los datos presentan una pendiente ascendente en su representación, indicando que los últimos problemas son más complejos de resolver. Esto entra en relación con lo visto anteriormente, cuando se hizo referencia a que ROLLER sin bagging era incapaz de resolver los últimos problemas propuestos.

En cuanto a las diferencias entre casos, destacan dos principalmente. Por un lado, se ve cómo los mejores tiempos los consigue el caso de 3 bags con el algoritmo de agregación (TBAP). Por otro lado, se ve cómo los peores tiempos los consiguen los experimentos de 10 y 20 bags. El uso de ROLLER sin bagging presenta resultados intermedios, necesitando en algunos casos más tiempo que otros algoritmos, y menos tiempo en otros.

Pese a ello, no hay demasiadas diferencias entre unos tiempos y otros para un mismo problema.

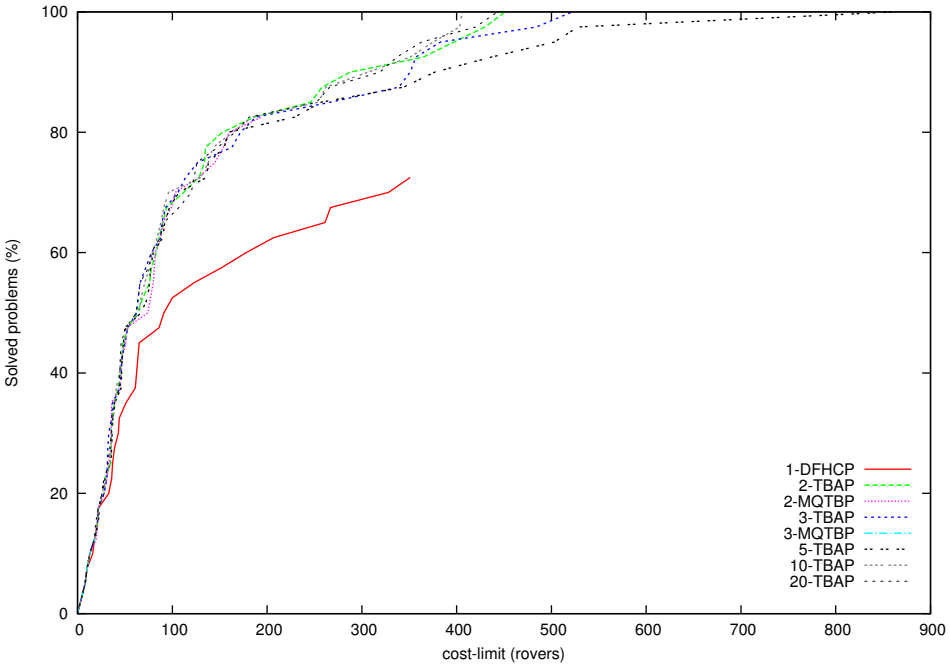

Figura 4.20: Límite de coste / Porcentaje de problemas resueltos - Rovers.

En la gráfica 4.20 se observa lo que se ha venido indicando con las gráficas anteriores.

En primer lugar, se ve cómo los algoritmos que consiguen un mayor porcentaje de problemas resueltos son los que emplean agregación: el 2-TBAP, 3-TBAP, 5-TBAP, 10-TBAP y 5-TBAP, alcanzando en los cinco casos el 100 % de problemas resueltos. Además, para este dominio se ve como en los tres primeros casos, a menor número de bags, los planes generados para alcanzar esa tasa presentan menor coste máximo; lo que sin duda es positivo, puesto que probablemente los tiempos dedicados serán también menores, al haber menos árboles que manejar. Por otro lado, los casos de 10 y 20 bags, alcanzan el 100 % con un menor coste, pero ya se ha visto que para ello requieren más tiempo, por lo que será necesario evaluar en cada caso si esto es rentable o no.

El siguiente caso más destacable es el 2-MQTBP, que sigue una línea muy similar a la de los tres casos anteriores. Con la diferencia de que ahora apenas se consigue superar el 80 % de problemas resueltos en su punto máximo.

Se ve también cómo el caso 3-MQTBP no ha conseguido tener mucho éxito, puesto que no consigue resolver el 20 % de los problemas asignados. Eso sí, los que resuelve, tienen un coste reducido.

Por último, el caso de ROLLER sin bagging no parece ser muy positivo, puesto que nunca supera el 70 % de problemas resueltos, alcanzando además

este punto para costes con los que los demás algoritmos han sido capaces de generar planes para un 85 %-90 % de los problemas planteados para este dominio.

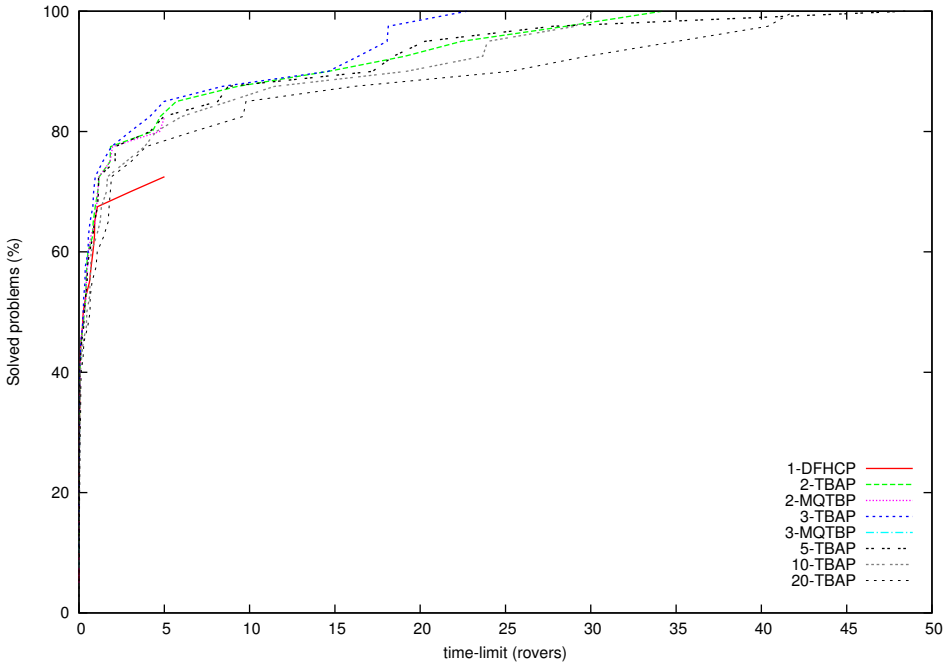

Figura 4.21: Límite de tiempo / Porcentaje de problemas resueltos - Rovers.

La gráfica 4.21 muestra cómo en la mayor parte de los casos se resuelve en torno al 65-70 % de los problemas propuestos en menos de 2 segundos, excepto el caso 3-MQTBP que ya se ha comentado que no consigue superar el 20 % en ningún caso.

A partir de ahí, se comienza a apreciar la diferencia entre los casos evaluados. Así, para ROLLER sin bagging (1-DFHCP), aunque el tiempo máximo no supere los 7'5 segundos, si se compara con el caso 2-MQTBP, sale perdiendo; puesto que éste último caso consigue alcanzar su máximo porcentaje de problemas resueltos en el mismo tiempo, siendo ese valor de en torno al 83 %.

Por otro lado, como ya se ha comentado, los casos que emplean el algoritmo TBAP, consiguen alcanzar el 100 % de problemas resueltos, aunque en tiempos mucho más elevados. Un buen valor, en torno al 90 %, se consigue en unos 15 segundos. Para aumentarlo hasta el 100 % es necesario, en algún caso, llegar casi a 50 segundos, por lo que puede ser rentable detenerse en el caso anterior, que tiene un porcentaje muy bueno; el cual, aunque no sea óptimo, se alcanza en un tiempo mucho menor en proporción.

4.3.5. Satellite

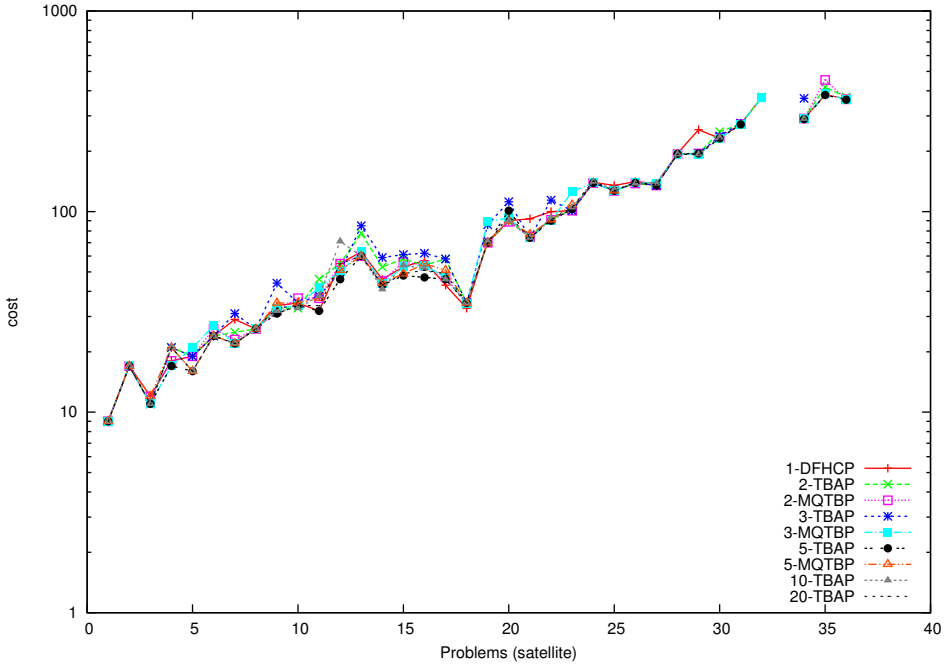

Figura 4.22: Problema / Coste (versión logarítmica)- Satellite.

La figura 4.22 muestra cómo el coste de los problemas va aumentando, lo que indica que los primeros problemas son más sencillos que los últimos.

Respecto a las diferencias entre cado caso, no son demasiado significativas a primera vista, generando planes con costes idénticos, especialmente en los primeros problemas.

Se puede ver cómo los costes para el caso 3-TBAP son ligeramente superiores al resto en algunos casos.

Otro dato destacable que se puede observar en la gráfica es que los últimos problemas no consiguen ser resueltos por ninguna de las aproximaciones. Además, el 3-TBAP no resuelve dos de los tres últimos problemas que el resto de casos sí resuelven.

Por último, hay que mencionar que esta vez, de nuevo, sí se consigue resolver problemas con 5 bags y el algoritmo MQTBP.

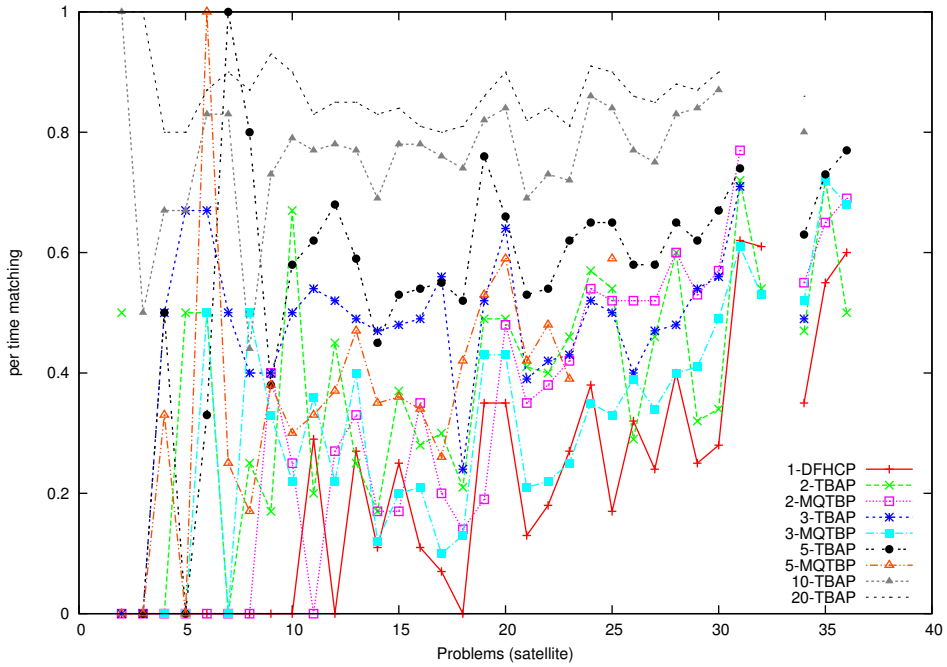

Figura 4.23: Problema / Relación de tiempo de matching entre tiempo total - Satellite.

En la gráfica 4.23 se puede apreciar una vez más, al igual que ocurría con el resto de dominios, cómo el ratio entre tiempo de matching y tiempo total es, por lo general, mayor para aquellos casos que emplean un mayor número de bags: 10-TBAP y 20-TBAP.

De forma análoga, los menores ratios son los que presenta el caso de ROLLER sin bagging.

El resto de casos muestran una tendencia algo variable, ya que para unos problemas se muestran superiores a otros casos, mientras que con otros problemas presentan unos ratios inferiores.

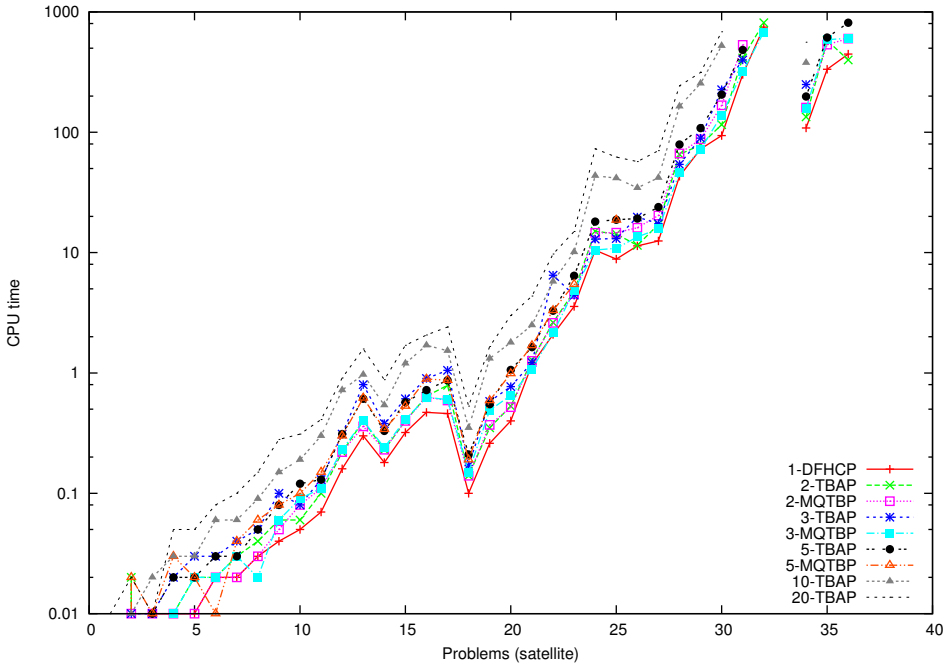

Figura 4.24: Problema / Tiempo de CPU (versión logarítmica) - Satellite.

La gráfica 4.24 muestra los tiempos de resolución de cada problema. Entra en relación con la primera gráfica, demostrando que a mayor coste de los planes generados, mayor es el tiempo dedicado por el planificador.

Se aprecia la tendencia ascendente, que indica una mayor dificultad para los últimos problemas del conjunto, como ya se había comentado.

Para este dominio, los menores tiempos los que muestra el caso de ROLLER sin bagging, mientras que los mayores tiempos los presentan los casos con un mayor número de bags, como el 10-TBAP o el 20-TBAP.

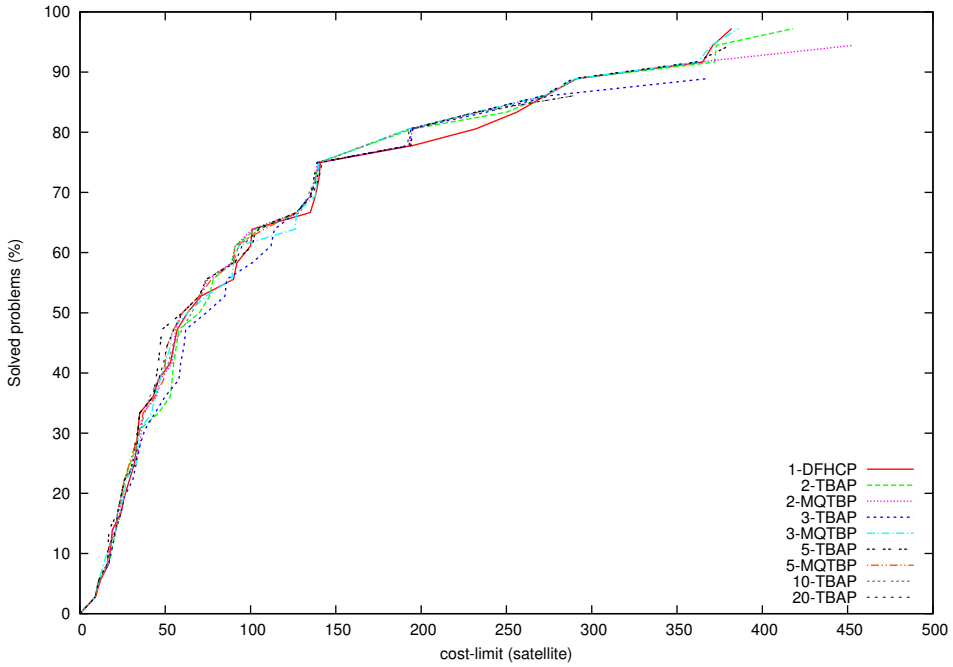

Figura 4.25: Límite de coste / Porcentaje de problemas resueltos - Satellite.

En la gráfica 4.25 se aprecia la dificultad de los problemas de este dominio, ya que ninguno de los casos probados consigue resolver el 100 % de los problemas.

En cuanto a las diferencias entre los casos probados, no son suficientemente significativas como para afirmar rotundamente la superioridad de uno de ellos, puesto que la tendencia mostrada por los datos es muy similar en todos los casos. Se podría mencionar que el caso 3-MQTBP y el 1-DFHCP consiguen los mayores porcentajes de éxito, junto con el 2-TBAP y el 2-MQTBP, sin embargo los primeros lo hacen generando planes de menor coste que los de los últimos.

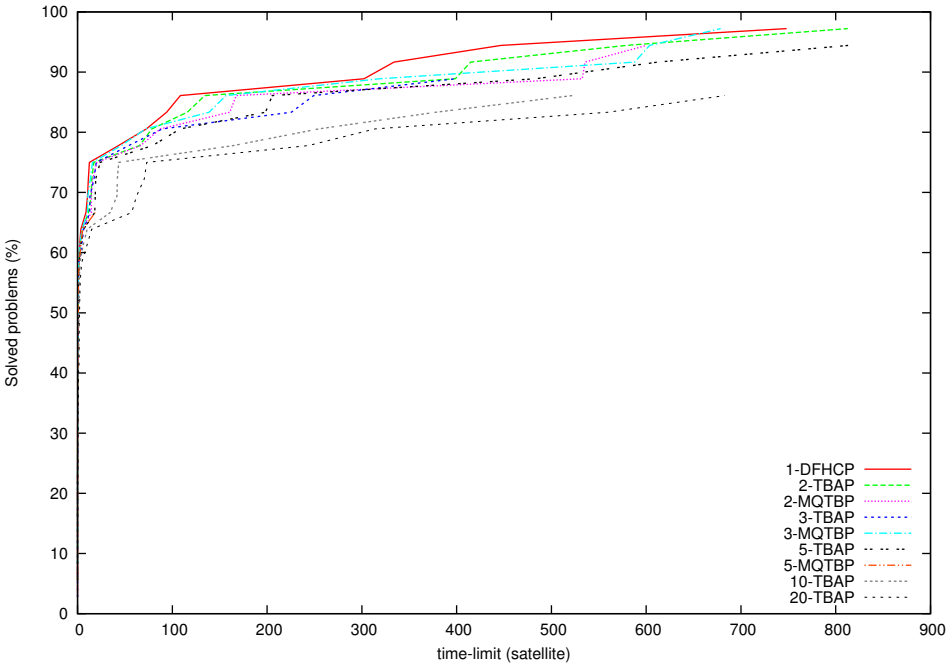

Figura 4.26: Límite de tiempo / Porcentaje de problemas resueltos - Satellite.

En relación con lo mencionado en los párrafos anteriores, la gráfica 4.26 muestra la dificultad del dominio Satellite. Se ve a simple vista cómo para este dominio el eje horizontal de la gráfica llega hasta el límite propuesto, 900 segundos, siendo el primer dominio de los probados en el que esto ocurre.

En el resto de dominios, los problemas que se conseguían resolver, lo hacían por norma general en menos de 100 segundos.

Sin embargo, se ve cómo estos tiempos son necesarios para conseguir los porcentajes más altos de problemas resueltos. Aunque, si el objetivo fuera tener un equilibrio entre un porcentaje de problemas resueltos adecuado y un tiempo dedicado no demasiado elevado, puede que sea preferible detener la ejecución antes de llegar al límite.

A la vista de los tiempos, esta idea se reafirma, puesto que se hace necesario un gran incremento del tiempo, para conseguir una mejora pequeña en el porcentaje de problemas resueltos.

Porcentajes de en torno al 75 % se obtienen en menos de 30 segundos para prácticamente la totalidad de los casos evaluados, por lo que una vez más sería buena idea plantear para este dominio detener la ejecución del planificador para tiempos muy superiores, en pos de una mayor eficiencia, al aprovechar mejor el tiempo, a pesar de resolver un menor porcentaje de problemas.

4.4. Resumen de problemas resueltos

Resumen: En las siguientes tablas se muestran de forma agrupada cuántos problemas han sido resueltos para cada una de las pruebas realizadas, de manera que sirva como comparativa.

4.4.1. Problemas resueltos por dominio

En la tabla 4.1 se muestra que el algoritmo que más problemas ha logrado resolver ha sido el caso de 5 bags con el algoritmo de agregación, resolviendo el 100 %. Le sigue de cerca, con 29/30 problemas resueltos el otro caso de 5 bags, con múltiples listas.

Los casos de 3 y 10 bags no han dado muy buenos resultados. Por otro lado, el uso de ROLLER sin bagging ha sido positivo, al conseguir resolver un gran número de problemas.

Número de bags	Algoritmo	Problemas resueltos
1	DFHCP	25/30
2	TBAP	26/30
	MQTBP	10/30
3	TBAP	15/30
	MQTBP	18/30
5	TBAP	**30/30**
	MQTBP	29/30
10	TBAP	2/30
20	TBAP	24/30

Tabla 4.1: Problemas resueltos para Blocksworld.

En la tabla 4.2 se muestra cómo para este dominio, la mayoría de los casos evaluados han respondido de manera similar, resolviendo entre 18 y 22 problemas de un total de 30. Esto puede indicar que existen determinados problemas dentro del conjunto de partida cuya dificultad ha hecho imposible generar un plan dentro del tiempo máximo permitido.

De hecho, los tres casos que mayor número de problemas han resuelto, presentan el mismo valor: 22/30. Lo que se podría interpretar como el límite máximo alcanzable por estas aproximaciones, existiendo 8 problemas dentro del conjunto inicial para lo que no es factible encontrar solución con los recursos empleados.

Número de bags	Algoritmo	Problemas resueltos
1	DFHCP	**22/30**
2	TBAP	21/30
	MQTBP	18/30
3	TBAP	**22/30**
	MQTBP	16/30
5	TBAP	19/30
	MQTBP	0/30
10	TBAP	**22/30**
20	TBAP	21/30

Tabla 4.2: Problemas resueltos para Depots.

La tabla 4.3 muestra que el conjunto de problemas empleado es, para este dominio, más asequible que en el caso anterior. Esto se puede ver por el hecho de que todos los experimentos evaluados han resuelto entre 26 y 28 problemas de 30.

Cinco de los nueve casos evaluados han alcanzado ese valor máximo: 26/28.

Número de bags	Algoritmo	Problemas resueltos
1	DFHCP	**28/30**
2	TBAP	27/30
	MQTBP	**28/30**
3	TBAP	26/30
	MQTBP	27/30
5	TBAP	**28/30**
	MQTBP	27/30
10	TBAP	**28/30**
20	TBAP	**28/30**

Tabla 4.3: Problemas resueltos para Parking.

La tabla 4.4 muestra resultados diversos.

En primer lugar, se ve el gran éxito del algoritmo de agregación (TBAP), que ha conseguido resolver **todos** los problemas para cada una de las aproximaciones realizadas.

En segundo lugar, en contraste con lo anterior, hay que mencionar que el otro algoritmo implementado ha presentado resultados mucho peores. Exceptuando el caso de 2 bags, que ha resuelto 34/40 problemas, el resto de pruebas (con 3 y 5 bags) apenas han podido resolver unos pocos problemas (5 problemas para el caso de 3 bags y ninguno para el de 5 bags).

Por último, el caso de ROLLER sin bagging, ha resuelto la mayoría de los casos, pero aún así el porcentaje de éxito no ha sido tan bueno como los obtenidos para el resto de pruebas. De esta manera, el caso DFHCP ha sido el tercer caso que menos problemas ha resuelto.

Número de bags	**Algoritmo**	**Problemas resueltos**
1	DFHCP	29/40
2	TBAP	**40/40**
	MQTBP	34/40
3	TBAP	**40/40**
	MQTBP	5/40
5	TBAP	**40/40**
	MQTBP	0/40
10	TBAP	**40/40**
20	TBAP	**40/40**

Tabla 4.4: Problemas resueltos para Rovers.

En la tabla 4.5 se muestran unos resultados similares para todos los casos.

Así, la mayoría de los casos evaluados han resuelto entre 31 y 35 problemas de un total de 36, lo que es bastante positivo. Las mayores tasas son presentadas (35/36) por los casos DFHCP y 2-TBAP, seguidos por 2-MQTBP y 3-MQTBP. En este caso, el planificador ha funcionado muy bien para el caso de 2 bags, ya que ambos algoritmos han salido bien parados. No se puede decir lo mismo de los casos de 5 bags, ya que un algoritmo ha resuelto 9 problemas menos que el otro, consiguiendo así el mínimo de todas las evaluaciones (24/36).

Número de bags	**Algoritmo**	**Problemas resueltos**
1	DFHCP	**35/36**
2	TBAP	**35/36**
	MQTBP	34/36
3	TBAP	32/36
	MQTBP	34/36
5	TBAP	33/36
	MQTBP	24/36
10	TBAP	31/36
20	TBAP	31/36

Tabla 4.5: Problemas resueltos para Satellite.

4.4.2. Problemas resueltos por algoritmo

En la tabla 4.6 se muestra cómo el algoritmo DFHCP ha funcionado mejor al aplicarse sobre los dominios Rovers y Parking, presentando resultados positivos en el resto de los casos.

Dominio	**Problemas resueltos**
Blocksworld	25/30
Depots	22/30
Parking	28/30
Rovers	29/40
Satellite	35/36

Tabla 4.6: Problemas resueltos por el algoritmo DFHCP (sin bagging).

En la tabla 4.7 se muestra cómo el algoritmo DFHCP ha funcionado mejor al aplicarse sobre el dominio Rovers, para el cual ha obtenido un 100 % de problemas resueltos para todos y cada uno de los casos evaluados, independientemente del número de bags empleado.

Más dispares son los resultados de Blocksworld, donde se pasa de un rotundo 30/30 con 5 bags, a tan solo 2/30 al usar 10 bags.

Para el resto de dominios, este algoritmo ha funcionado de manera muy positiva, resolviendo la mayoría de problemas y, además, haciéndolo para todas las evaluaciones, con distinto número de bags.

Además, otro punto a favor de este algoritmo es el hecho de que haya sido capaz de resolver problemas para los casos de 10 y 20 bags, cosa que no ha ocurrido, como se verá a continuación (ver tabla 4.8), con el algoritmo MQTBP.

En la tabla 4.8 se muestra cómo el algoritmo DFHCP ha funcionado mejor al aplicarse sobre el dominio Parking.

Por contra, en Rovers los resultados han sido bastante malos para dos de los tres casos evaluados, a diferencia de lo que ocurría con el algoritmo TBAP que consiguió resolver todos.

Para Satellite se puede decir lo contrario, ya que dos de los tres casos evaluados han resuelto casi todos los problemas planteados.

Por último, para los dominios restantes, el algoritmo MQTBP no ha conseguido en conjunto buenos resultados, pese a que pueda destacar en alguno de los casos evaluados; como, por ejemplo, el caso de Blocksworld con 5 bags, para el que se han resuelto 29/30 problemas.

Dominio	**Número de bags**	**Problemas resueltos**
Blocksworld	2	26/30
	3	15/30
	5	30/30
	10	2/30
	20	24/30
Depots	2	21/30
	3	22/30
	5	19/30
	10	22/30
	20	21/30
Parking	2	27/30
	3	26/30
	5	28/30
	10	28/30
	20	28/30
Rovers	2	40/40
	3	40/40
	5	40/40
	10	40/40
	20	40/40
Satellite	2	35/36
	3	32/36
	5	33/36
	10	31/36
	20	31/36

Tabla 4.7: Problemas resueltos por el algoritmo TBAP (agregación).

Dominio	Número de bags	Problemas resueltos
Blocksworld	2	10/30
	3	18/30
	5	29/30
Depots	2	18/30
	3	16/30
	5	0/30
Parking	2	28/30
	3	27/30
	5	27/30
Rovers	2	34/40
	3	5/40
	5	0/40
Satellite	2	34/36
	3	34/36
	5	24/36

Tabla 4.8: Problemas resueltos por el algoritmo MQTBP (múltiples listas).

Capítulo 5

Gestión del proyecto

Resumen: En este capítulo se van a comentar algunos aspectos relevantes de cómo se ha gestionado el proyecto actual. En especial, se enfatizará en la planificación temporal realizada, mostrando un resumen de las tareas llevadas a cabo y las fechas en las que éstas se han desarrollado. Finalmente, se detallará el presupuesto estimado del proyecto.

5.1. Fases del desarrollo del proyecto

El presente proyecto ha sido la conclusión de meses de trabajo, durante los cuales se han ido desarrollando diferentes aspectos para dar forma a una idea inicial, hasta llegar a su puesta en marcha. Para ello ha sido necesario no sólo implementar el sistema requerido, sino todo un esfuerzo previo de investigación sobre las bases de partida, que ofreciesen un soporte teórico y práctico sobre el que empezar a construir la nueva versión. Igualmente, el proyecto no finaliza con la generación del código, sino que para llegar a las conclusiones expuestas en este documento ha sido necesaria una gran inversión de tiempo en la realización de diferentes baterías de pruebas, con las cuales se esperaba generar un conjunto de resultados satisfactorio del que extraer información relevante.

Con la finalidad de esclarecer todo el proceso realizado para finalizar el proyecto, desde su inicio, en este apartado se detallarán las diferentes tareas propuestas y el orden en el que se han ido realizando, de manera que se pueda tener una idea clara de todos los aspectos que han sido tenidos en cuenta.

De forma general, el proyecto puede ser dividido en varias fases diferenciadas entre sí. Éstas son:

1. **Adquisición de información:** conocer las bases teóricas del proyecto,

aprender los lenguages de programación a utilizar y manejar el software de partida.

2. **Preparación del entorno:** descarga, instalación y compilación de programas, paquetes y librerías.
3. **Desarrollo:** generación del código nuevo.
4. **Experimentación:** realización de las baterías de pruebas. Generación de resultados y post-procesado de datos.
5. **Documentación:** aprendizaje de LaTeX, creación de plantilla y documento, revisiones e impresión final.
6. **Presentación:** fase final de exposición del proyecto.

La imagen 5.1 muestra la lista completa de tareas y subtareas:

Id	Nombre de tarea	Duración	Comienzo	Fin
1	Elección del proyecto	2 horas	jue 27/09/12	jue 27/09/12
2	Inicio del proyecto	0 días	jue 27/09/12	jue 27/09/12
3	**Adquisición de información**	**4,63 días**	**vie 28/09/12**	**mar 02/10/12**
4	**De ROLLER**	**2,63 días**	**dom 30/09/12**	**mar 02/10/12**
5	Lectura de la documentación	2 horas	dom 30/09/12	dom 30/09/12
6	Estudio de la estructura de clases	2 días	dom 30/09/12	mar 02/10/12
7	Manejo básico de la aplicación	3 horas	mar 02/10/12	mar 02/10/12
8	**De Python**	**3 días**	**vie 28/09/12**	**dom 30/09/12**
9	Lectura de manuales	2 días	vie 28/09/12	sáb 29/09/12
10	Programación de ejercicios de aprendizaje	1 día	dom 30/09/12	dom 30/09/12
11	**Preparación del entorno**	**0,19 días**	**lun 01/10/12**	**lun 01/10/12**
12	Descarga e instalación de ROLLER	30 mins	lun 01/10/12	lun 01/10/12
13	Descarga e instalación de ACE	30 mins	lun 01/10/12	lun 01/10/12
14	Descarga e instalación de paquetes auxiliares	10 mins	lun 01/10/12	lun 01/10/12
15	Instalación del repositorio de compartición	30 mins	lun 01/10/12	lun 01/10/12
16	Familiarización con el repositorio	1 hora	lun 01/10/12	lun 01/10/12
17	**Desarrollo**	**46 días**	**jue 04/10/12**	**dom 18/11/12**
18	**Programa roller_trainer**	**17,1 días**	**jue 04/10/12**	**dom 21/10/12**
19	Generar ficheros de configuración de los árboles	9 días	jue 04/10/12	vie 12/10/12
20	Comprobación de ficheros de configuración de los árboles creados	20 mins	sáb 13/10/12	sáb 13/10/12
21	Generar ficheros de entrenamiento	6 días	lun 15/10/12	sáb 20/10/12
22	Comprobación de ficheros de entrenamiento creados	20 mins	dom 21/10/12	dom 21/10/12
23	Llamada a TILDE	10 mins	dom 21/10/12	dom 21/10/12
24	Comprobación de los árboles creados	20 mins	dom 21/10/12	dom 21/10/12
25	**Ampliación de ROLLER**	**19 días**	**lun 22/10/12**	**vie 09/11/12**
26	Lectura y almacenamiento de varios árboles	4 días	lun 22/10/12	jue 25/10/12
27	Modificación del algoritmo DFHCP para actuar sobre estructura creadas	6 días	vie 26/10/12	mié 31/10/12
28	Creación del algoritmo TBAP	8 días	vie 02/11/12	vie 09/11/12
29	Creación del algoritmo MQTBP	8 días	vie 02/11/12	vie 09/11/12
30	**Pruebas de todo el proceso anterior**	**7 días**	**lun 12/11/12**	**dom 18/11/12**
31	Sin bagging	7 días	lun 12/11/12	dom 18/11/12
32	Con bagging y algoritmo TBAP	7 días	lun 12/11/12	dom 18/11/12
33	Con bagging y algoritmo MQTBP	7 días	lun 12/11/12	dom 18/11/12
34	**Desarrollo del programa main_script**	**3 días**	**sáb 10/11/12**	**lun 12/11/12**
35	Entrenamiento para un caso	1 día	sáb 10/11/12	sáb 10/11/12
36	Validación para un caso	1 día	dom 11/11/12	dom 11/11/12
37	Entrenamiento para varios casos a la vez	2 días	dom 11/11/12	lun 12/11/12
38	Validación para varios casos a la vez	1 día	lun 12/11/12	lun 12/11/12
39	Pruebas de main_script	2 horas	lun 12/11/12	lun 12/11/12
40	**Experimentación**	**84 días**	**jue 22/11/12**	**mié 13/02/13**
41	Realización de primera tanda de experimentos por SSH	8 días	jue 22/11/12	jue 29/11/12
42	**Modificaciones del código**	**1 día**	**lun 03/12/12**	**lun 03/12/12**
43	Alternativa a subprocess usando threads	1 día	lun 03/12/12	lun 03/12/12
44	Cambio de servidor	0 horas	mar 04/12/12	mar 04/12/12
45	Realización de segunda tanda de experimentos por SSH	9 días	jue 20/12/12	vie 28/12/12
46	**Tratamiento de resultados**	**3 días**	**jue 24/01/13**	**sáb 26/01/13**
47	Instalación de programa de creación de tablas	1 hora	jue 24/01/13	jue 24/01/13
48	Creación de script intermedio para formatear los datos	3 días	jue 24/01/13	sáb 26/01/13
49	Generación de las tablas	2 horas	sáb 26/01/13	sáb 26/01/13
50	**Realización de tercera tanda de experimentos por SSH**	**8 días**	**mié 06/02/13**	**mié 13/02/13**
51	Creación de tablas de tercera tanda	8 días	mié 06/02/13	mié 13/02/13
52	**Documentación**	**191 días**	**jue 06/12/12**	**vie 14/06/13**
53	Aprendizaje de LaTeX	7 días	jue 06/12/12	mié 12/12/12
54	Creación de plantilla	7 días	jue 13/12/12	mié 19/12/12
55	Creación del documento	77 días	vie 01/02/13	jue 18/04/13
56	Primera revisión del documento	17 días	vie 19/04/13	dom 05/05/13
57	Primera corrección	13 días	lun 06/05/13	sáb 18/05/13
58	Segunda revisión del documento	6 días	sáb 18/05/13	jue 23/05/13
59	Segunda corrección	18 días	vie 24/05/13	lun 10/06/13
60	Impresión de la documentación	4 días	mar 11/06/13	vie 14/06/13
61	Creación de presentación	3 días	mar 11/06/13	jue 13/06/13
62	Presentación del proyecto	1 hora	lun 24/06/13	lun 24/06/13
63	Fin del proyecto	0 días	lun 24/06/13	lun 24/06/13

Figura 5.1: Lista de tareas y subtareas del proyecto.

La imagen 5.2 presenta un diagrama de Gantt, para las tareas principales:

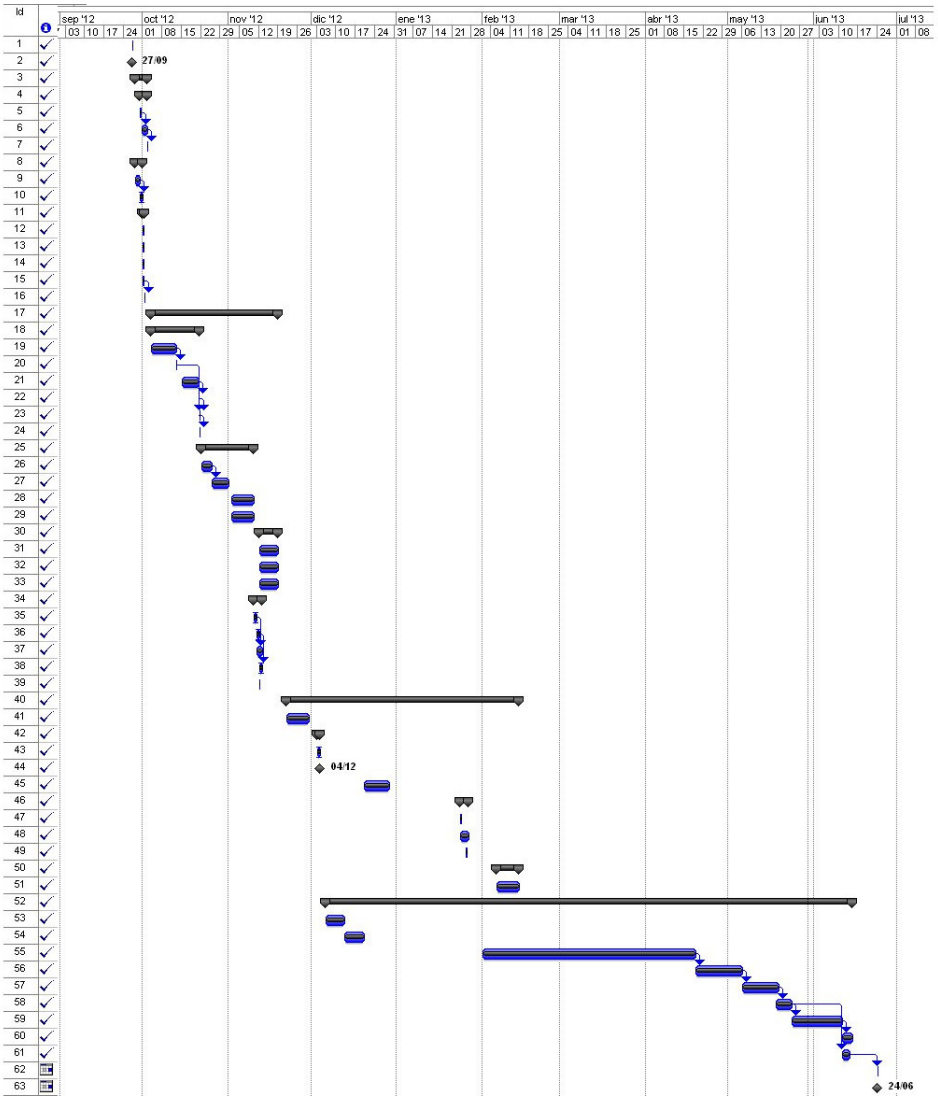

Figura 5.2: Diagrama de Gantt.

5.2. Presupuesto

En esta sección se detalla el presupuesto del proyecto, en base a determinados criterios sugeridos por la UC3M. Para algunas de las tablas, se detallan las fórmulas utilizadas que permiten obtener los valores de la columna correspondiente.

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor:
Alberto Garbajosa Poderoso
2.- Departamento:
Departamento de Informática
3.- Descripción del Proyecto:
- Titulo: **Generación de Políticas para Planificación Heurística mediante Metaclasificadores**
- Duración (meses): **9**
Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):
33.605,78 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación [a] (hombres mes)	Coste hombre mes	Coste (Euro)
de la Rosa Turbides, Tomás		Ingeniero Senior	1	4.289,54	4.289,54
Fuentetaja Pizán, Raquel		Ingeniero Senior	1	4.289,54	4.289,54
Garbajosa Poderoso, Alberto		Ingeniero	7	2.694,39	18.860,73
		Hombres mes 9		**Total**	27.439,81

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

Figura 5.3: Detalles del proyecto y costes de personal.

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Servidor 4 núcleos Intel Xeon 3GHz	899,00	100	2	60	29,97
Intel Core2 Duo 2'4GHz	720,00	60	7	60	51,84
				Total	81,81

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} \text{xCxD}$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

Figura 5.4: Coste de equipos informáticos.

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
No aplica	-	-
	Total	0,00

Figura 5.5: Subcontratación de tareas.

OTROS COSTES DIRECTOS DEL PROYECTO[e)]

Descripción	Empresa	Costes imputable
Dietas	*Varias*	180,00
Electricidad	Iberdrola	130,00
Línea ADSL	Movistar	270,00
Material fungible	*Varias*	10,00
Transporte	EMT de Madrid	73,20
	Total	483,20

[e)] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

Figura 5.6: Otros costes.

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	27.440
Amortización	82
Subcontratación de tareas	0
Costes de funcionamiento	483
Costes Indirectos	5.601
Total	33.606

Figura 5.7: Resumen de costes.

Capítulo 6

Conclusiones

Con la realización de este proyecto se pretendía investigar vías alternativas de planificación automática, a partir de proyectos anteriores. De esta manera, el objetivo fundamental era comprobar cómo se comportaba un sistema de planificación heurística, que emplea como soporte a la decisión árboles de decisión generados por medio de técnicas de aprendizaje automático.

Proyectos anteriores mostraban buenos resultados con la inclusión de un árbol de decisión. Ahora, se ha querido comprobar si mediante políticas de combinación de varios árboles, esta tendencia positiva continuaba o, por el contrario, no se veía una alternativa clara.

Tras la realización del proyecto se puede afirmar que se han cumplido todos los objetivos propuestos.

- En primer lugar, se ha conseguido plantear e implementar la manera de separar un conjunto de entrenamiento inicial en varios conjuntos, empleando para ello técnicas de bagging.

- Posteriormente, se ha implementado el código necesario para ampliar el sistema de partida, ROLLER, de manera que permita la generación de varios árboles de decisión relacionales a partir de los distintos conjuntos de entrenamiento y la manipulación de dichos árboles, empleando para ello dos aproximaciones diferentes: una de ellas suma las decisiones de cada uno de los árboles y genera una lista de nodos ordenada; la otra, modifica el algoritmo de búsqueda de manera que escoge alternativamente la información de un solo árbol cada vez, empleando una lista abierta de nodos distinta para cada árbol (bag).

- Por último, tras un amplio proceso de experimentación, se ha podido probar el planificador para diferentes problemas de dominios distintos bajo determinadas condiciones. Así, tras las pruebas realizadas, se ha podido observar cómo, en primer lugar, el uso de varios árboles de decisión simultaneamente no supone una mejora evidente al uso de un

sólo árbol. Por otro lado, tampoco se puede deducir que su uso empeore las pruebas realizadas hasta ahora.

De este modo, tras la realización de los experimentos se pueden extraer algunas conclusiones, que se exponen a continuación.

En primer lugar, se ha observado cómo el uso de técnicas de bagging ha supuesto en muchos casos resolver problemas que no se resuelven con el uso de un sólo árbol de decisión. Si bien es cierto que las pruebas realizadas no han sido concluyentes en cuanto al número óptimo de agrupaciones (*bags*) con el que se obtienen mejores resultados de manera generalizada, sí se puede decir que en la mayoría de los casos existía un número de grupos con el que se conseguía resolver un problema para el que técnicas sin bagging no conseguían generar ningún plan.

Además, de manera similar, los tiempos de resolución de problemas eran iguales o mejores al usar bagging respecto a los experimentos que no lo usaban en muchas de las pruebas realizadas.

Aunque, de nuevo, esto no es así para todos los casos de bagging. Se ha podido observar cierta tendencia a empeorar los resultados con un número muy bajo de grupos (dos, por ejemplo). Esto puede ser debido a la aleatoriedad con la que se asignan los problemas en bagging, que dejan problemas fuera del conjunto de entrenamiento, haciendo que los árboles generados no posean la suficiente información.

Al incluir más agrupaciones, este efecto se atenúa, puesto que de manera general los problemas que no aparecen en el conjunto asignado a un grupo, podrán aparecer en otro, y cuantos más grupos haya mayor es la probabilidad de que todos los problemas aparezcan en alguno de los casos. Por ello, al combinar la información de varios árboles, para un número más grande de agrupaciones se obtienen mejores resultados.

Respecto a esto último, se ha podido observar cierta tendencia que muestra que de los dos algoritmos implementados, el que suma la información de todos los árboles presenta resultados ligeramente mejores (en coste y tiempo) a los generados por el algoritmo que usa los árboles alternativamente, para casos con pocas agrupaciones. Si se observan los experimentos realizados con muchos grupos, esta diferencia es abismal, ya que mientras que el primer algoritmo ha dado solución a la mayoría de los problemas que se le planteaban, el segundo algoritmo no ha conseguido resolver ni un solo problema para casos de 10 bags en adelante en el tiempo asignado.

En cuanto a algunos aspectos negativos de usar varios árboles en lugar de uno solo, el primero de ellos es evidente, y es que se ha de contar con mayor espacio de almacenamiento para poder generar cada uno de los árboles. A pesar de que no es una diferencia demasiado significativa para una sola prueba, si se realizan a la vez muchas pruebas con diversos dominios, la capacidad de almacenamiento de la máquina que se esté usando será un dato a tener en cuenta.

En segundo lugar, está el uso de memoria principal, que también ha de ser mayor que antes para poder cargar en memoria todos los árboles que se quieran usar al mismo tiempo.

Por último, respecto al tiempo empleado, hay que hacer una pequeña distinción. Por un lado se encuentra el tiempo de entrenamiento, que ahora con bagging es, evidentemente, mayor; puesto que en vez de entrenar para crear un sólo árbol de decisión, habrá que hacer este proceso varias veces. Indiscutiblemente a mayor número de agrupaciones mayor será el tiempo que se requiera para generar cada uno de los árboles necesarios. Por otro lado, el tiempo de validación se ha observado que es similar al empleado con un sólo árbol. Así, para algunos casos se mejoran los tiempos, y para otros se empeora, pero no hay diferencias realmente significativas.

El hecho de que el sistema permita separar el entrenamiento de la validación, hace que lo expuesto en el párrafo anterior no sea realmente crítico de cara a la resolución de problemas, ya que una vez invertido el tiempo en crear los árboles, se podrá hacer la validación tantas veces como se considere oportuno, empleando distintos algoritmos, modificando los tiempos máximos por problema o realizando experimentos únicamente para algunos de los dominios entrenados. De este modo, de forma práctica, si se llegara a un punto en el que se han conseguido buenos árboles, capaces de resolver un determinado tipo de problemas, no será necesario generar nuevos árboles, dedicando el tiempo disponible únicamente a las pruebas, siendo este tiempo similar al empleado en ejecuciones sin bagging.

Quizás, como aspecto negativo de cara a la comparativa entre pruebas, está el hecho de no haber podido generar resultados con más de 10 grupos para uno de los algoritmos. A pesar de que en alguna ocasión se consiguieron resolver algunos problemas y con buenos planes, en la mayoría de los casos el tiempo de validación llegaba al máximo permitido, rechazando dichos problemas. Esto hace que en la práctica no sea demasiado útil usar un número muy alto de agrupaciones con esta implementación del algoritmo, dado que sale demasiado caro entrenar todos los árboles para apenas solucionar unos pocos problemas.

En referencia a lo anterior, en el siguiente capítulo (*Capítulo 7*) se comentarán algunas vías de investigación futuras a partir del proyecto actual, que entre otras cosas permitan ver qué ocurriría con un número de grupos muy grande, con la finalidad de tener una nueva visión comparativa de la eficiencia de los algoritmos.

En conclusión, con el proyecto actual se ha podido demostrar que el uso de metaclasificadores como soporte a la planificación heurística, no supone siempre una mejora trivial a la generación de planes de manera automática respecto al uso de un único árbol de decisión, sino que distintas aproximaciones presentan resultados variados que hacen necesario plantear la manera de abordar los problemas propuestos, con el fin de obtener los mejores planes

posibles en un tiempo razonable.

Capítulo 7

Trabajos futuros

Resumen: En este capítulo se van a comentar algunas propuestas para futuras líneas de continuación de este proyecto, como realizar pruebas con más grupos (bags), generar el conjunto de entrenamiento de otras maneras o emplear nuevos algoritmos de combinación de los árboles.

A la vista de los resultados de las pruebas y de las conclusiones que de ellas han sido extraídas en los capítulos anteriores, se pueden plantear ciertas visiones alternativas de los procesos realizados que ayuden a conseguir mejores resultados o que permitan explorar nuevas vías de investigación no tratadas en este proyecto.

Algunas propuestas en este sentido se comentan en las siguientes secciones.

7.1. Pruebas con más grupos

Para este proyecto se realizaron pruebas con hasta 20 agrupaciones, sin embargo resultó que en la mayoría de los casos a partir de 8 ó 9 grupos el tiempo máximo asignado era insuficiente para que pudieran generarse resultados con uno de los algoritmos implementados. (Por esta razón no se han mostrado en el capítulo de Experimentación).

Sería interesante ver qué planes se generan con un número de agrupaciones grande. Para ello, podría verse como alternativa bien aumentar el tiempo máximo por problema o bien emplear una máquina más potente que fuera capaz de resolver los problemas en ese tiempo.

En referencia a esto último, se realizó alguna prueba aislada, revelando que empleando otras máquinas para las pruebas, para un mismo problema bajo las mismas condiciones (número de grupos, dominio, tiempos máximos, etcétera) se conseguía reducir el tiempo empleado en resolverlo a la mitad.

De esto se podría deducir que quizás sí se consiguieran resolver problemas con un número más elevado de grupos; pero no ha sido posible realizar más pruebas, por lo que aquí se presenta como propuesta.

7.2. Maneras alternativas de generar el conjunto de entrenamiento

El conjunto de entrenamiento es generado escogiendo tantos problemas como haya en el conjunto, pero de manera aleatoria.

Esto quiere decir que, si en el conjunto inicial se presentan 30 problemas, el conjunto de entrenamiento empleado en bagging será también de 30 problemas. Sin embargo, al haberse escogido al azar, puede haber problemas repetidos y, por lo tanto, problemas que permanezcan sin usar.

Teniendo en cuenta que éste es el principio de todo el proceso, un cambio radical en el conjunto de entrenamiento puede afectar a los árboles de decisión generados y, por consiguiente, a la capacidad de decisión del planificador.

En base a esto, podrían plantearse diferentes alternativas de generación del conjunto de entrenamiento, con el fin de evaluar hasta qué punto puede afectar o no la selección de los problemas. Por ejemplo:

- Sin selección aleatoria: establecer varios conjuntos manualmente.
- Establecer mecanismos para asegurar que un mismo problema no es escogido más de dos veces, para garantizar cierta variabilidad.
- Escoger la mitad del conjunto de forma fija y la otra mitad de manera aleatoria.

7.3. Nuevos algoritmos de combinación

En este proyecto se han propuesto dos algoritmos para combinar los árboles de decisión creados. En el primero de ellos, se suman las decisiones de todos los árboles. En el segundo, se escogen de forma alternativa las decisiones de cada árbol.

Investigar nuevas posibilidades de combinar la información que contienen los árboles de decisión, puede dar lugar a algoritmos más eficientes.

Por ejemplo, se podría implementar un método de votación, en el que se escoja la salida mayoritaria de entre todas las propuestas por los diferentes clasificadores construidos. Éste es el método típico empleado por bagging para tareas de clasificación.

Otra ejemplo relacionado podría ser modificar no sólo los algoritmos de combinación, sino los de creación. Para generar los árboles de decisión se

ha usado en este proyecto una herramienta externa, ACE. Cualquier avance llevado a cabo en la generación de clasificadores, por esta u otra herramienta, podría repercutir positivamente en la calidad de los planes generados al utilizarlos. Incluso se podría plantear el uso de diferentes herramientas para cada uno de los clasificadores, lo que les dotaría de mayor variedad.

Bibliografía

Blockeel, H. y De Raedt, L. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, (101), 1998.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J. y Vandecasteele, H. Executing query packs in ILP. En *Inductive Logic Programming, 10th International Conference, ILP2000, London, UK, July 2000, Proceedings*, vol. 1866 de *Lecture Notes in Artificial Intelligence*, páginas 60–77. Springer, 2000. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=31608.

Breiman, L. *Bagging predictors*. 24. 1996.

Hoffmann, J. y Nebel, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, vol. 14, páginas 253–302, 2001.

Quinlan, J. R. *C4.5 : programs for machine learning*. Morgan Kaufmann Publishers, Inc, 1993. ISBN 1-55860-238-0.

Richter, S. y Westphal, M. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, (39), páginas 127–177, 2010.

de la Rosa, T., Jiménez, S., Fuentetaja, R. y Borrajo, D. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research*, (40), 2011.

Lista de acrónimos

BAGGING ..... Bootstrap AGGregatING

DFHCP ..... Depth-First H-Context Policy

MQTBP ..... Multiple Queue Tree Bagging Policy

NASA ..... National Aeronautics and Space Administration

PDDL ..... Planning Domain Definition Language

PLG ..... Planning & Learning research Group

TBAP ..... Tree-Bagging Aggregation Policy

UC3M ..... Universidad Carlos III de Madrid

Apéndice A

Manual de usuario

Resumen: En este capítulo se va a explicar de forma breve cómo hacer funcionar el sistema, indicando cuál debe ser la estructura de directorios requerida, cómo compilar ROLLER y cómo ejecutar las pruebas para generar resultados.

A.1. Consideraciones iniciales

Este sistema ha sido diseñado teniendo en mente los resultados, por encima de otros factores como su usabilidad final. Con su realización se ha tratado de investigar una hipótesis mediante la realización de diferentes experimentos. Es por ello que el software actual no ha sido pensado para ser manipulado inicialmente por usuarios inexpertos, sino que para su correcto funcionamiento se requerirán unos mínimos conocimientos técnicos, especialmente para hacerlo funcionar por primera vez.

De este modo, podrá ser necesario organizar los diferentes programas de que consta el sistema, compilar alguno de ellos o modificar los permisos de ejecución. Igualmente, la modificación de determinados parámetros del sistema, requerirá la manipulación del código fuente.

Asimismo, es recomendable poseer conocimientos básicos de planificación automática y aprendizaje automático, para entender qué implicaciones puede tener el modificar determinados parámetros del sistema, como por ejemplo el número de grupos al utilizar algoritmos de bagging.

Una vez aclarado esto, en los siguientes apartados se detallarán todos estos aspectos, de manera que se facilite el proceso de ejecución de las pruebas deseadas.

A.2. Requisitos iniciales

Este sistema emplea diferentes programas para su funcionamiento, implementados en distintos lenguajes de programación y que necesitan determinados requisitos previos para poder funcionar correctamente.

De forma general, para este sistema se requiere:

- Linux.
- Compilador C. *El sistema se ha probado con la versión 4.7.2 de gcc.*
- Python. *El sistema se ha probado con la versión 2.7.3.*
- Programa ACE versión 1.2.20.

Asimismo, podrá ser necesario instalar determinadas bibliotecas y/o paquetes de software para algunas funciones auxiliares del sistema.

A.3. Estructura de directorios

En muchos de los casos, al leer ficheros, el sistema emplea rutas absolutas, pudiendo así encontrar lo que busca sin problemas. Pero este hecho no ha sido una prioridad durante el desarrollo, por lo que es posible que para determinados casos se estén empleando rutas relativas, pudiendo afectar al buen funcionamiento del sistema si no se emplea la misma estructura que aquí se indica.

Es por ello que, en este manual, se explicará la estructura de directorios empleada durante todo el desarrollo y pruebas del sistema, de manera que se asegure su correcto funcionamiento.

Dicho esto, en primer lugar se contará con una carpeta raíz a elección del usuario. Dentro de ella, tendrá que haber la siguiente estructura de carpetas:

- ACE-1.2.20: contiene la versión 1.2.20 del programa ACE.
- domains: contiene los dominios. Cada uno de ellos se encontrará en su propio directorio. Para cada uno habrá, inicialmente:
  - Fichero de dominio .pddl.
  - Directorio "probsets": contiene todos los problemas que se van a usar, tanto los de entrenamiento como los de validación.
- Roller: contiene todos los ficheros de ROLLER.
- Trainer: contiene:
  - main_script.py (requiere permiso de ejecución).

- roller_trainer.py (requiere permiso de ejecución).
- Directorio "pddl": contiene un programa para extraer la información de los ficheros .pddl.

A.4. Compilación y preparación del sistema

Antes de ejecutar, es necesario compilar ROLLER. Para ello, hay que desplazarse hasta el directorio Roller y, una vez dentro, ejecutar el comando:

```
make
```

Además, es necesario comprobar que los ficheros "main_script.py" y "roller_trainer.py" poseen permiso de ejecución.

A.5. Ejecución del sistema

Para ejecutar main_script, hay que situarse en el directorio correspondiente (*(...)/Trainer/*), y escribir la llamada:

```
./main_script.py
```

Esa llamada ofrecerá un recordatorio de los parámetros de entrada, por lo que una nueva llamada a main_script.py seguida de los parámetros oportunos comenzará la batería de pruebas predeterminada. *NOTA: como ya se comentó en el apartado de Script de automatización de las pruebas 3.3.2, es posible modificar el código para que las pruebas se hagan bajo condiciones diferentes.*

Una llamada de ejemplo es:

```
./main_script.py ../domains/ 120 1 0
```

Con esta llamada anterior estamos diciendo que los dominios se encuentran en la carpeta "domains", que se encuentra un nivel por encima del directorio actual, que se quiere hacer entrenamiento con un tiempo máximo de 120 segundos por problema y que no se quiere hacer validación.

A.6. Resultados de las pruebas

Si se ha seguido la estructura indicada, el sistema almacenará los resultados de las pruebas en los directorios de dominio.

De esta manera, cada dominio probado tendrá:

- Directorio "results", dentro del cual habrá:

- Ficheros de resultados. B.6
- Ficheros de problemas rechazados. B.7

Además, hay que recordar que dentro de cada carpeta de dominio se habrán creado los ficheros intermedios, necesarios para el planificador (ficheros .s B.2, .kb B.3 y árboles B.4).

Apéndice B

Formato de los ficheros

Resumen: Este apéndice muestra el formato que tienen los ficheros que participan del proceso de planificación, tanto los que sirven como entrada al sistema, como los que son generados por él como salida.

B.1. Fichero de dominio

Contiene la definición del dominio en lenguaje PDDL. Presenta, así, los tipos de elementos que van a participar, los predicados y los operadores.

Ejemplo para el dominio blocksworld (*blocksworld.pddl*):

```
(define (domain blocksworld)

(:requirements :strips :typing)
(:types block)
(:predicates (on ?x - block ?y - block)
(ontable ?x - block)
(clear ?x - block)
(handempty)
(holding ?x - block)
)

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

```
(holding ?x)))

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

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

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

B.2. Fichero de configuración de los árboles (.s)

Los ficheros .s se crean a partir de los ficheros de dominio.

Cuando se quieren generar los árboles de decisión para un determinado bag, el primer paso es extraer la información del dominio, y dividirla en los ficheros .s, existiendo uno de ellos para elegir la mejor acción, y uno por cada una de las acciones del dominio, para elegir su mejor instanciación.

La estructura de estos ficheros tiene cuatro partes diferenciadas:

1. Opciones de TILDE.

2. Objetivos.

3. Predicados.

4. Comandos de ACE.

A continuación se pone un ejemplo de un fichero .s correspondiente a la acción *pick-up* del dominio *Blocksworld*:

```
%%*****************************************
%% File Automatically generated
%% To learn the selection of nodes
%% LEARNING pick-up EPISODES
%%*****************************************
% Tilde Options
tilde_mode(classify).
typed_language(yes).
minimal_cases(5).
output_options([c45,c45c,c45e,lp,prolog_probab]).

% The target concept
predict(selected_pick_up(+Example,+Exprob,+BLOCK0,-Class)).
type(selected_pick_up(example,exprob,block,class)).
classes([selected,rejected]).

rmode(target_goal_on(+Example,+Exprob,+-A,+-B)).
type(target_goal_on(example,exprob,block,block)).

rmode(static_fact_on(+Exprob,+-A,+-B)).
type(static_fact_on(exprob,block,block)).

rmode(target_goal_ontable(+Example,+Exprob,+-A)).
type(target_goal_ontable(example,exprob,block)).

rmode(static_fact_ontable(+Exprob,+-A)).
type(static_fact_ontable(exprob,block)).

rmode(target_goal_clear(+Example,+Exprob,+-A)).
type(target_goal_clear(example,exprob,block)).

rmode(static_fact_clear(+Exprob,+-A)).
type(static_fact_clear(exprob,block)).

rmode(target_goal_handempty(+Example,+Exprob)).
type(target_goal_handempty(example,exprob)).
```

```
rmode(static_fact_handempty(+Exprob)).
type(static_fact_handempty(exprob)).

rmode(target_goal_holding(+Example,+Exprob,+-A)).
type(target_goal_holding(example,exprob,block)).

rmode(static_fact_holding(+Exprob,+-A)).
type(static_fact_holding(exprob,block)).

% The domain predicates
rmode(helpful_pick_up(+Example,+Exprob,+-A)).
type(helpful_pick_up(example,exprob,block)).

rmode(helpful_put_down(+Example,+Exprob,+-A)).
type(helpful_put_down(example,exprob,block)).

rmode(helpful_stack(+Example,+Exprob,+-A,+-B)).
type(helpful_stack(example,exprob,block,block)).

rmode(helpful_unstack(+Example,+Exprob,+-A,+-B)).
type(helpful_unstack(example,exprob,block,block)).

rmode(nothelpful_pick_up(+Example,+Exprob,+-A)).
type(nothelpful_pick_up(example,exprob,block)).

rmode(nothelpful_put_down(+Example,+Exprob,+-A)).
type(nothelpful_put_down(example,exprob,block)).

rmode(nothelpful_stack(+Example,+Exprob,+-A,+-B)).
type(nothelpful_stack(example,exprob,block,block)).

rmode(nothelpful_unstack(+Example,+Exprob,+-A,+-B)).
type(nothelpful_unstack(example,exprob,block,block)).

% The ACE commands
execute(tilde).
execute(quit).
```

B.3. Fichero de entrenamiento (.kb)

Los ficheros .kb se obtienen a partir de los ficheros .s, existiendo el mismo número de ficheros de un tipo que del otro. Son generados en base a las

instrucciones que contenían los ficheros .s.

Los ficheros .kb sirven como entrada a ACE para generar los árboles de decisión con TILDE, por lo que están estructurados de manera que puedan ser entendidos por ese sistema.

Son muy extensos, ya que contienen la información de todos los ejemplos de entrenamiento.

A continuación se muestra un fragmento de un fichero .kb para la acción *pick-up* del dominio *Blocksworld*. El fragmento se corresponde a la primera parte del fichero, en la que se muestra la información de un ejemeplo de entrenamiento. Después irían los demás ejemplos hasta completar todo el conjunto empleado.

```
% TRAINING EXAMPLES FROM FF-LEARNER (ROLLER)
% Static Predicates of problem

% Example train_ensemble05_12_046_E1
selected_pick_up(train_ensemble05_12_046_E1,train_ensemble05_12_046,b10,rejected).
state_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11).
state_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b10).
state_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b12,b3).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11,b7).
state_ontable(train_ensemble05_12_046_E1,train_ensemble05_12_046,b10).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9,b1).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8,b6).
state_ontable(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6,b4).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5,b12).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4,b5).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3,b2).
state_ontable(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
state_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1,b8).
state_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11,b1).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9,b8).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8,b5).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7,b9).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5,b11).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4,b2).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3,b10).
target_goal_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2,b3).
achieved_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6,b4).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
```

```
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b12).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b12).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11).
rpadd_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1).
rpadd_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11).
rpadd_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b10).
rpdel_ontable(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
```

```
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3,b2).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b12).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b12,b3).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5,b12).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4,b5).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6,b4).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8,b6).
rpdel_holding(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1,b8).
rpdel_ontable(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9,b1).
rpdel_clear(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11).
rpdel_handempty(train_ensemble05_12_046_E1,train_ensemble05_12_046).
rpdel_on(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11,b7).
helpful_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11,b7).
helpful_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9,b1).
nothelpful_pick_up(train_ensemble05_12_046_E1,train_ensemble05_12_046,b10).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2,b3).
```

```
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4,b2).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3,b10).
rpaction_pick_up(train_ensemble05_12_046_E1,train_ensemble05_12_046,b2).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b3,b2).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5,b11).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b12,b3).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8,b5).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b5,b12).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b4,b5).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b6,b4).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b9,b8).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7,b9).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b8,b6).
rpaction_stack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b11,b1).
rpaction_unstack(train_ensemble05_12_046_E1,train_ensemble05_12_046,b1,b8).
rpaction_pick_up(train_ensemble05_12_046_E1,train_ensemble05_12_046,b7).

% Example train_ensemble05_12_046_E2
state_clear(train_ensemble05_12_046_E2,train_ensemble05_12_046,b11).
state_clear(train_ensemble05_12_046_E2,train_ensemble05_12_046,b10).
```

Continúa hasta completar todos los problemas...

B.4. Fichero de árbol de decisión

Contiene una representación del árbol de decisión generado para un grupo (*bag*).

A continuación se muestra un ejemplo de fichero de árbol de decisión para la acción *pick-up* del dominio *Blocksworld*:

```
selected_pick_up(-A,-B,-C,-D)

target_goal_on(A,B,C,-E) ?
+-yes: nothelpful_unstack(A,B,E,-F) ?
| +-yes: [selected] 81.0 [[selected:81.0,rejected:0.0]]
| +-no: nothelpful_pick_up(A,B,E) ?
| +-yes: target_goal_on(A,B,-G,C) ?
| | +-yes: target_goal_on(A,B,-H,G) ?
| | | +-yes: [selected] 19.0 [[selected:17.0,rejected:2.0]]
| | | +-no: [selected] 7.0 [[selected:6.0,rejected:1.0]]
| | +-no: [selected] 12.0 [[selected:12.0,rejected:0.0]]
```

```
| +-no: [rejected] 261.0 [[selected:0.0,rejected:261.0]]
+-no: [rejected] 223.0 [[selected:0.0,rejected:223.0]]
```

B.5. Fichero de problemas asignados

Cuando se quiere probar un algoritmo de bagging, se generan tantos grupos (*bags*) como se indique al programa. Por cada grupo, se generará un fichero de problemas asignados, que contendrá los ficheros que se han escogido para entrenar ese grupo.

El fichero de problemas asignados tendrá tantos problemas como total de ellos haya en el conjunto de problemas inicial. Sin embargo, estos se escogen de manera aleatoria, pudiendo estar repetidos.

En cada línea del fichero se presenta la ruta relativa de cada problema asignado y, a su lado, si se ha aceptado en el entrenamiento ("OK") o se ha rechazado por agotar el tiempo máximo permitido ("REJECTED").

A continuación se muestra un fragmento de ejemplo de fichero de problemas asignados a un caso de prueba de 2 bags aplicado al dominio *blocksworld* (*problemas_ asignados.txt*):

```
../domains/blocksworld/probsets/train-ensemble03-10-021.pddl OK

../domains/blocksworld/probsets/train-ensemble04-11-037.pddl OK

../domains/blocksworld/probsets/train-ensemble04-11-033.pddl REJECTED

../domains/blocksworld/probsets/train-ensemble05-12-048.pddl OK

Continúa hasta completar el número de problemas correspondiente...
```

B.6. Fichero de resultados

Contiene los resultados de todos los problemas resueltos para una prueba, consistente en aplicar un algoritmo a un conjunto de problemas de un dominio, para un número determinado de bags.

En la primera línea aparecen los nombres de los valores que se van a mostrar. En la siguiente, el nombre del dominio. Estas dos líneas aparecen comentadas (empiezan por el símbolo "#"), de manera que no interfieran en el caso de que se quiera automatizar el procesado de los datos (por ejemplo, para generar las tablas que se han visto en el capítulo de experimentación).

Después, en cada línea, se presentan los resultados de un problema. En caso de que un problema no haya sido resuelto, no se deja ningún espacio.

Es decir, si se han resuelto los problemas 3 y 5, pero no el 4, en la línea siguiente al tercer problema irá la línea del quinto problema.

Cada una de las líneas muestra, en orden de izquierda a derecha, los valores indicados en la primera línea del fichero, como ya se ha comentado. Muchos de estos valores presentan información relevante para algunos procesos experimentales de ROLLER, y no han sido utilizados para este proyecto, pero se han mantenido por simplicidad. En cualquier caso, pueden ser modificados en el código de ROLLER. Por defecto son los siguientes:

- Nombre del problema.
- Coste total.
- Número de operadores.
- Número de estados evaluados.
- Profundidad máxima.
- Tiempo de búsqueda (segundos).
- *Valor irrelevante para el proyecto*
- Tiempo total.
- *Valor irrelevante para el proyecto*
- Máximo número de elementos en la lista abierta.
- Relación entre tiempo de matching y tiempo total.

A continuación se muestra un fragmento de ejemplo para el dominio *Blocksworld*, empleando el algoritmo *TBAP* (número 31) con 2 bags (*results_2bags_31.txt*):

```
# fct_file_name total-cost ops-number
number-of-evaluated-states max-depth-ehc seconds-searching
second-building-relaxed-graphplan seconds-total-time
total-relaxed-graphplan-levels max-open-list-elements
matching-time/seconds-total-time

# ../domains/blocksworld/blocksworld.pddl

../domains/blocksworld/probsets/ipc7-learn002.pddl 236.00 236 237
0 0.97 0.62 1.03 1991 0.11

../domains/blocksworld/probsets/ipc7-learn003.pddl 224.00 224 225
```

```
0 0.90 0.60 0.97 2787 0.06

../domains/blocksworld/probsets/ipc7-learn004.pddl 182.00 182 183
0 0.75 0.44 0.82 1622 0.12
```

Continúa hasta completar el número de problemas resueltos...

B.7. Fichero de problemas rechazados

Contiene una lista con los problemas que han sido rechazados para una prueba, consistente en aplicar un algoritmo a un conjunto de problemas de un dominio, para un número determinado de bags.

En cada línea se muestra un problema rechazado en esa prueba (su ruta relativa). Los problemas que no aparecen aquí han sido resueltos dentro del tiempo máximo asignado.

A continuación se muestra un ejemplo para el dominio *Blocksworld*, empleando el algoritmo *TBAP* (número 31) con 2 bags (*rejected_ 2bags_ 31.txt*):

```
../domains/blocksworld/probsets/ipc7-learn023.pddl

../domains/blocksworld/probsets/ipc7-learn030.pddl
```

Apéndice C

Detalle de la formalización de los dominios

Resumen: Este apéndice muestra los tipos, predicados y acciones de los que se compone cada uno de los dominios probados.

C.1. Blocksworld

La definición de este dominio empleada en el proyecto consta de:

1. Tipos: 1.
   a) Bloque.
2. Predicados: 5.
   a) Bloque encima de otro.
   b) Bloque en la mesa.
   c) Bloque libre.
   d) Brazo libre.
   e) Brazo sujetando un bloque.
3. Acciones: 4.
   a) Coger bloque.
   b) Dejar bloque en la mesa.
   c) Apilar bloque sobre otro.
   d) Desapilar bloque.

Las pruebas cuyos resultados se han mostrado en el capítulo de experimentación 4.3.1, se corresponden a:

1. Conjunto de entrenamiento: 50 problemas.
2. Conjunto de validación: 30 problemas.

C.2. Depots

La definición de este dominio empleada en el proyecto consta de:

1. Tipos: 9.
   a) Lugar (objeto).
   b) Objeto localizable (objeto).
   c) Depósito (lugar).
   d) Distribuidor (lugar).
   e) Camión (localizable).
   f) Montacargas (localizable).
   g) Superficie (localizable).
   h) Palé (superficie).
   i) Caja (superficie).
2. Predicados: 6.
   a) Objeto localizable en un lugar.
   b) Caja en una superficie.
   c) Caja dentro de un camión.
   d) Montacargas levantando caja.
   e) Montacargas disponible.
   f) Superficie despejada.
3. Acciones: 5.
   a) Conducir camión de un lugar a otro lugar.
   b) Levantar una caja.
   c) Dejar una caja.
   d) Cargar el camión.
   e) Descargar el camión.

Las pruebas cuyos resultados se han mostrado en el capítulo de experimentación 4.3.2, se corresponden a:

1. Conjunto de entrenamiento: 50 problemas.
2. Conjunto de validación: 30 problemas.

C.3. Parking

La definición de este dominio empleada en el proyecto consta de:

1. Tipos: 2.
    a) Coche.
    b) Bordillo.
2. Predicados: 5.
    a) Coche en bordillo.
    b) Coche en número de bordillo.
    c) Coche detrás de otro.
    d) Coche libre.
    e) Bordillo libre.
3. Acciones: 4.
    a) Mover coche de bordillo a bordillo.
    b) Mover coche de bordillo a doble fila.
    c) Mover coche de doble fila a bordillo.
    d) Mover coche de doble fila a doble fila.

Las pruebas cuyos resultados se han mostrado en el capítulo de experimentación 4.3.3, se corresponden a:

1. Conjunto de entrenamiento: 50 problemas.
2. Conjunto de validación: 30 problemas.

C.4. Rovers

La definición de este dominio empleada en el proyecto consta de:

1. Tipos: 7.
    a) Rover.
    b) Posición.
    c) Almacén.
    d) Cámara.
    e) Modo.
    f) Módulo de aterrizaje.

g) Objective.

2. Predicados: 25.

   a) Rover en posición.
   b) Módulo de aterrizaje en posición.
   c) Puede ir rover de una posición a otra posición.
   d) Rover equipado para análisis de tierra.
   e) Rover equipado para análisis de roca.
   f) Rover equipado para romar imágenes.
   g) Almacén vacío.
   h) Rover tiene muestra de roca de una posición.
   i) Rover tiene muestra de tierra de una posición.
   j) Almacén lleno.
   k) Rover tiene cámara calibrada.
   l) Cámara admite modo.
   m) Rover disponible.
   n) Posición visible desde otra posición.
   ñ) Rover tiene imagen de objetivo en modo.
   o) Transferido datos de tierra de una posición.
   p) Transferido datos de roca de una posición.
   q) Transferido datos de imagen de una posición.
   r) Muestra de tierra en posición.
   s) Muestra de roca en posición.
   t) Objetivo visible desde posición.
   u) Almacén pertenece a rover.
   v) Cámara calibrada con objetivo.
   w) Cámara en rover.
   x) Canal libre en módulo de aterrizaje.

3. Acciones: 9.

   a) Mover rover de una posición a otra.
   b) Tomar muestra de tierra.
   c) Tomar muestra de roca.
   d) Soltar muestras en almacén.
   e) Calibrar cámara.

f) Tomar imagen.

g) Transferir datos de tierra.

h) Transferir datos de roca.

i) Transferir datos de imagen.

Las pruebas cuyos resultados se han mostrado en el capítulo de experimentación 4.3.4, se corresponden a:

1. Conjunto de entrenamiento: 50 problemas.
2. Conjunto de validación: 40 problemas.

C.5. Satellite

La definición de este dominio empleada en el proyecto consta de:

1. Tipos: 4.
   a) Satélite.
   b) Dirección.
   c) Instrumento.
   d) Modo.
2. Predicados: 8.
   a) Saber qué instrumento está en el satélite.
   b) Los modos que soporta cada instrumento.
   c) Hacia qué dirección apunta el satélite.
   d) Si está disponible el satélite.
   e) Si está encendido el instrumento.
   f) Si está calibrado el instrumento.
   g) Si existe una imagen en una dirección y un modo.
   h) La dirección de calibración de un instrumento.
3. Acciones: 5.
   a) Hacer que un satélite apunte en una dirección.
   b) Activar un instrumento.
   c) Desactivar un instrumento.
   d) Calibrar un instrumento de un satélite en una dirección.
   e) Tomar imagen, con un instrumento en un modo.

Las pruebas cuyos resultados se han mostrado en el capítulo de experimentación 4.3.5, se corresponden a:

1. Conjunto de entrenamiento: 50 problemas.
2. Conjunto de validación: 36 problemas.