Facultad de Matemática, Astronomía y Física

Universidad Nacional de Córdoba

Trabajo especial de la Licenciatura en Ciencias de la Computación

---

Optimización de dominios de Planning

Autor:
Raúl Ezequiel Medina Fabersani
ezemed7@gmail.com

Director:
Dr. Carlos Areces
carlos.areces@gmail.com

Codirector:
Dr. Martín Ariel Domínguez
mardom75@gmail.com

Clasificación (CCS 1998):

- D.1.4 PROGRAMMING TECHNIQUES: Sequential Programming
- I.2.9 ROBOTICS: Workcell organization and planning
- F.2.0 ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY

Palabras Claves:

- Planning
- Split
- Unsplit
- Dominios
- Optimización
- Rendimiento
- PDDL

Resumen: En este trabajo se describe una técnica de optimización de dominios de planning. Primero se presenta una introducción acerca de la inteligencia artificial en general. Luego se aborda el problema de planning revisando los momentos históricos más importantes hasta llegar al estado del arte. También se introduce el lenguaje PDDL para poder presentar nuevos dominios que se implementaron. Luego se presenta la técnica de optimización propuesta, la cual incluye el algoritmo de Split y Unsplit. Con el fin de poder estudiar el rendimiento de la técnica de optimización propuesta, se realizan pruebas de rendimiento sobre los dominios presentados y un dominio clásico. Por último se estudian los resultados obtenidos para poder dar una conclusión sobre la técnica en general.

Summary: This work explain an optimization technique for planning domains. First comes an introduction about artificial intelligence in general. Then describes the planning problem, the most important historical moments are reviewd until the state of the art. After that, the language PDDL is introduced, to understand the new domains implemented. Then the Split and Unsplit algorithms are presented. Finally after running tests, the results and conclusion are shown.

Índice general

Resumen En este trabajo se describe una técnica de optimización de dominios de planning. Primero se presenta una introducción acerca de la inteligencia artificial en general. Luego se aborda el problema de planning revisando los momentos históricos más importantes hasta llegar al estado del arte. También se introduce el lenguaje PDDL para poder presentar nuevos dominios que se implementaron. Luego se presenta la técnica de optimización propuesta, la cual incluye el algoritmo de Split y Unsplit. Con el fin de poder estudiar el rendimiento de la técnica de optimización propuesta, se realizan pruebas de rendimiento sobre los dominios presentados y un dominio clásico. Por último se estudian los resultados obtenidos para poder dar una conclusión sobre la técnica en general.

1. Introducción

En esta tesis vamos a trabajar en el área de *Planning* la cual es una subárea de la *Inteligencia Artificial* (IA). Actualmente existen muchas definiciones sobre que es la IA repartidas en distintos enfoques filosóficos. Tomaremos una definición de cada enfoque para tener una idea más completa [1].

Sistemas que piensan como humanos. "La automatización de actividades que asociamos con humanos pensando, actividades como toma de decisiones, resolución de problemas, aprendizaje, ..." (Bellman, 1978).

Sistemas que actúan como humanos. "El arte de crear máquinas que realizan funciones que requieren inteligencia cuando son realizadas por personas" (Kruzweil, 1990).

Sistemas que piensan racionalmente. "El estudio de facultades mentales a través del uso de modelos computacionales" (Charniak and McDermott, 1985).

Sistemas que actúan racionalmente. "El área de estudio que busca explicar y emular el comportamiento inteligente en términos de procesos computacionales" (Schalkoff, 1990).

En nuestro caso la IA nos sirve para resolver problemas generales, no necesariamente de la forma óptima pero si con gran flexibilidad.

Un *agente inteligente* que intenta realizar una tarea, debe tomar decisiones sobre las acciones que realizará en cada paso durante la ejecución. También existen varios enfoques sobre como implementar este tipo de "inteligencia".

Enfoque basado en programación: En este enfoque el programador es quien resuelve los problemas y prescribe los movimientos posibles en cada situación escribiendo un programa o una colección de reglas de comportamiento. Claramente este enfoque es muy frágil ya que el programador puede no prever todos los comportamientos posibles, y cualquier situación no contemplada es un problema que el agente no podrá resolver.

Enfoque basado en aprendizaje: En este enfoque las decisiones sobre las acciones a ejecutar las toma un programa controlador. Este programa va "aprendiendo" las mejores decisiones tomadas a través del tiempo (experiencia), y utiliza esta experiencia en las nuevas tomas de decisiones.

Este enfoque funciona muy bien ya que potencialmente el agente toma cada vez mejores decisiones. Pero, es posible que el agente aprenda sobre uno o varios ambientes particulares y las decisiones aprendidas quedan muy ligadas a estos ambientes, limitando la capacidad de decidir sobre un ambiente desconocido de forma correcta.

Enfoque basado en modelos: En este enfoque el controlador de decisiones es derivado automáticamente a partir de un modelo de acciones, sensores, estados, eventos y metas.

Un sistema basado en modelos debe respetar algunas restricciones. Debe ser un *sistema finito*, es decir que cuenta con un número finito de estados, acciones y eventos. Debe ser *determinístico*, es decir que cada acción tiene un solo resultado. Debe tener *metas alcanzables*, es decir que debe existir un conjunto de estados de llegada. Debe generar *planes secuenciales*, es decir que los planes obtenidos deben ser una secuencia de acciones ordenadas que nos permitan llegar a la meta. Debe tener el *tiempo implícito*, es decir que no puede tener acciones que que se ejecuten durante alguna cantidad de tiempo, sino que todas las acciones se consideran instantáneas.

Este enfoque es utilizado en Planning Clásico. Pero, tiene la desventaja de ser muy costoso computacionalmente.

1.1. El problema de Planning

El problema de la planificación automática es el de generar una secuencia finita de acciones (llamados planes), para ser ejecutados por agentes inteligentes. Estas acciones deben llevar al agente desde un estado inicial hacia un objetivo dado llamado meta [2]. Se debe tener en cuenta que estos planes no siempre existen y también que pueden existir muchos planes que cumplan el mismo objetivo. Los algoritmos encargados de encontrar dichos planes se llaman planificadores.

Ya que los problemas que resuelve un planificador potencialmente no están limitados a problemas o dominios específicos, hay que considerar todo el espacio de búsqueda del problema para poder encontrar una solución. Se sufre por lo tanto el problema de la explosión combinatoria de estados al tratar de encontrar un plan. Si tenemos un número p de proposiciones primitivas que describen los posibles estados en que puede estar un agente, tendremos 2^p estados posibles. Más exactamente, la planificación pertenece al conjunto de problemas de complejidad PSPACE-complete [3]. Sin embargo existen algoritmos de planning que pueden resolver problemas de tamaño y complejidad considerables.

Ejemplo. Tomaremos como ejemplo al clásico dominio de la *International Planning Competition* (IPC) 2011 [4], llamado *Gripper*. En este dominio hay algunos

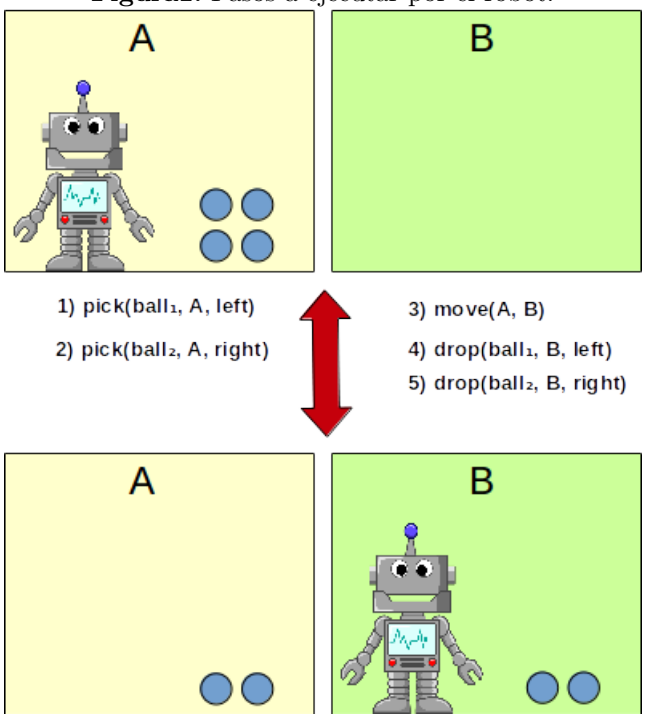
robots con dos pinzas cada uno, y cierta cantidad de habitaciones con pelotas. Los robots tienen la tarea de llevar las pelotas de unas habitaciones a otras.

El problema particular que trataremos de resolver consta de un solo robot, dos habitaciones y cuatro pelotas todas en la habitación A. Nuestro robot deberá llevar todas las pelotas a la habitación B.

El robot puede ejecutar sólo tres acciones. La acción *pick* sirve para levantar una pelota, y toma como parámetros a la pelota que va a levantar, la habitación donde se encuentra esa pelota y por último la pinza con la que lo hará. La acción *move* sirve para llevar al robot de una habitación a otra, y los parámetros que toma son los nombres de las dos habitaciones. Y la acción *drop* sirve para soltar la pelota en la habitación donde está parado el robot, toma como parámetros a la pelota que desea soltar, la habitación donde desea soltar la pelota, y por último la pinza con la que lo hará.

Utilizando estas acciones se debe encontrar un plan para resolver el problema que planteamos. Es fácil ver que el problema tiene solución, ya que el robot no tiene ningún tipo de limitación para moverse o levantar las pelotas, entonces podemos encontrar un plan simplemente ejecutando acciones como las de la figura.

Figura1. Pasos a ejecutar por el robot.



Siguiendo estos pasos podemos ver que un plan completo que resolvería nuestro problema es el siguiente.

```
pick(ball_1, A, left)
pick(ball_2, A, right)
move(A, B)
drop(ball_1, B, left)
drop(ball_2, B, right)
move(B, A)
pick(ball_3, A, left)
pick(ball_4, A, right)
move(A, B)
drop(ball_3, B, left)
drop(ball_4, B, right)
```

Lo que no es tan fácil de ver es que este es el plan óptimo para resolver nuestro problema. Si comparamos este plan con otro plan válido, donde el robot lleva las pelotas una por una.

```
pick(ball_1, A, left)
move(A, B)
drop(ball_1, B, left)
move(B, A)
pick(ball_2, A, left)
move(A, B)
drop(ball_2, B, left)
move(B, A)
pick(ball_3, A, left)
move(A, B)
drop(ball_3, B, left)
move(B, A)
pick(ball_4, A, left)
move(A, B)
drop(ball_4, B, left)
```

Podemos ver que el último plan es más largo y por lo tanto más costoso que el primero, en este problema. Para comprobar que el primer plan presentado es el óptimo deberíamos compararlo con todos los otros planes válidos que existen y verificar que efectivamente es el de menor costo.

De todas formas en esta tesis solo nos enfocaremos en encontrar planes que resuelvan nuestros problemas, sin importar que sean o no los planes óptimos.

A continuación mostraremos la codificación de este problema en lenguaje PDDL, un lenguaje de especificación de dominios de planning. En la sección 2.1 introduciremos PDDL formalmente.

```
(define (domain gripper-typed)
   (:requirements :typing)
```

```
(:types room ball gripper)
(:constants left right - gripper)
(:predicates (at-robby ?r - room)
   (at ?b - ball ?r - room)
   (free ?g - gripper)
   (carry ?o - ball ?g - gripper))

(:action move
    :parameters (?from ?to - room)
    :precondition (at-robby ?from)
    :effect (and (at-robby ?to)
        (not (at-robby ?from))))

(:action pick
    :parameters (?obj - ball ?room - room ?gripper - gripper)
    :precondition (and (at ?obj ?room) (at-robby ?room) (free
        ?gripper))
    :effect (and (carry ?obj ?gripper)
       (not (at ?obj ?room))
       (not (free ?gripper))))

(:action drop
    :parameters (?obj - ball ?room - room ?gripper - gripper)
    :precondition (and (carry ?obj ?gripper) (at-robby ?room))
    :effect (and (at ?obj ?room)
       (free ?gripper)
       (not (carry ?obj ?gripper)))))
```

El código anterior define lo que se llama el dominio de un problema de planning. Un dominio de planning describe las acciones que los agentes pueden realizar. En forma independiente, se especifican los problemas de planning para un dominio dado que indican los objetos existentes en el dominio, el estado inicial y la meta que se desea alcanzar.

```
(define (problem gripper4)
   (:domain gripper-typed)
   (:objects rooma roomb - room
            ball1 ball2 ball3 ball4 - ball)
   (:init (at-robby rooma)
          (free left)
          (free right)
          (at ball1 rooma)
          (at ball2 rooma)
          (at ball3 rooma)
          (at ball4 rooma))
   (:goal (and (at ball1 roomb)
```

```
            (at ball2 roomb)
            (at ball3 roomb)
            (at ball4 roomb))))
```

Al correr un planificador sobre este dominio y este problema obtenemos, por ejemplo, el siguiente plan.

```
step   0: pick ball1 rooma right
       1: pick ball2 rooma left
       2: move rooma roomb
       3: drop ball1 roomb right
       4: drop ball2 roomb left
       5: move roomb rooma
       6: pick ball3 rooma right
       7: pick ball4 rooma left
       8: move rooma roomb
       9: drop ball3 roomb right
      10: drop ball4 roomb left
```

En este caso el planificador nos retornó el plan óptimo. Como puede verse en este ejemplo, los lenguajes de especificación de tareas de planning son de alto nivel y permiten codificar situaciones complejas e interesantes.

1.2. Historia

El primer planificador y uno de los primeros programas que son considerados como pertenecientes al área de Inteligencia Artificial fue introducido por Newell and Simon en 1959 [5]. El programa se llamó *General Problem Solver* (GPS) e introduce una técnica llamada *means-ends analysis*, donde una acción es elegida si ésta reduce la diferencia entre el estado actual y la meta. Esta acción produce un nuevo estado, y este proceso es aplicado recursivamente hasta alcanzar la meta. El sistema STRIPS [6] combina means-ends analysis con un conveniente lenguaje declarativo. En años subsiguientes la técnica de means-ends analysis fue refinada y extendida en muchos aspectos, resultando en la formulación de algoritmos de planificación que son *sound* (solo producen planes), *complete* (produce un plan si existe uno) y *effective* (escala hasta grandes problemas). A principio de los 90 el estado del arte en planificadores era el UCPOP [7], una implementación de un elegante método de planificación, conocido como *partial-order* planning donde los planes no son buscados desde el estado inicial hacia adelante o desde la meta hacia atrás, sino que se construyen a partir de un esquema de descomposición en el que se descomponen los objetivos conjuntos. Las acciones se incorporan al plan parcial ordenadas como sea necesario con el fin de resolver conflictos entre ellos. Los algoritmos de orden parcial son *sound* y *complete* pero no escalan adecuadamente.

La situación en planning cambio drásticamente a mediados de los 90 con la introducción de *Graphplan* [8], un algoritmo que parecía tener poco en común con

los enfoques utilizados anteriormente, pero que escalaba mucho mejor. Graphplan construye un gráfico del plan en tiempo polinomial pensándolo hacia adelante a partir de un estado inicial, luego se busca desde la meta hacia atrás para encontrar un plan. Más tarde se demostró que la razón por la que Graphplan funciona tan bien es por una heurística implícita en el gráfico del plan.

El éxito de Graphplan impulsó otros enfoques. Por ejemplo el enfoque basado en SAT (i.e., satisfabilidad de fórmulas proposicionales), donde un problema de planificación particular es convertido en un problema general de *satisfabilidad* expresado como un conjunto de cláusulas en *Conjuntive Normal Form* (CNF) asumiendo que el plan, si existe, no tiene más de un número dado de pasos. Estas clausulas luego se resuelven utilizando un SAT-Solver (i.e., un razonador automático para lógica proposicional). Si estas clausulas son satisfactibles, cualquier valuación que las satisfaga genera un plan para el problema original. Si las cláusulas no son satisfactibles se intenta nuevamente elevando el número de pasos asumidos para el plan. Actualmente se puede resolver enormes instancias SAT, pero el problema de SAT es NP-Complete. En la actualidad la formulación de Planning Clásico que parece escalar mejor se basa en búsqueda heurística. La búsqueda heurística utiliza un *valor heurístico* precalculado como guía para decidir que camino seguir en el árbol de búsqueda. Los valores heurísticos se pueden obtener de muchas formas. La técnica más popular es llamada *delete-relaxation*, y consiste en resolver un dominio relajado, que no tiene acciones negativas (*deletes*). Luego de conseguir un plan relajado, el valor heurístico será el costo de este plan. En este momento se puede realizar la búsqueda heurística y obtener el plan real si este existe. Además de las heurísticas, los planificadores del estado del arte utilizan información sobre las acciones más útiles en cada estado para encontrar los planes.
Algunos planificadores también utilizan sub-objetivos implícitos en el problema llamados *landmarks*, estos estados son extraídos automáticamente del problema con métodos parecidos a los utilizados para derivar los valores heurísticos.

1.3. Planners

En la actualidad existen varios algoritmos de planificación. Nos enfocaremos en los planificadores de *satisficing planning*, los que buscan obtener algún plan si es que existe, sin importar si el plan es o no óptimo (i.e., el de menor cantidad de acciones, o el de menor costo).

Los planificadores son capaces de manejar diferentes lenguajes de especificación de acciones. Los lenguajes más conocidos son STRIPS y ADL. Ambos están basados en el lenguaje estándar de especificación de dominios llamado PDDL. A continuación presentaremos algunos de los planificadores más conocidos.

Fast Forward [9] Es un sistema de planificación independiente del dominio, desarrollado por el Dr. Jörg Hoffmann [10]. Es un planificador progresivo, es decir que busca en el espacio de estados a partir del estado inicial hacia adelante hasta alcanzar el objetivo.

Para decidir que rama del árbol de planificación debe elegir primero utiliza una estimación heurística basada en un gráfico de plan similar al utilizado en *Graphplan*.

Fast Forward es capaz de manejar especificaciones de acciones escritas en STRIPS y en ADL.

Fast Downward [11] Es un sistema de planificación clásico basado en búsqueda heurística. Puede tratar problemas generales de planificación determinista especificados en PDDL 2.2, incluyendo STRIPS, ADL y axiomas. Al igual que Fast Forward es un planificador progresivo. Fast Downward utiliza descomposiciones jerárquicas de tareas de planificación para el cálculo de la función heurística, denominada *Causal Graph Heuristic*. Es una heurística diferente a las clásicas que utilizan la delete-relaxation para obtener el valor heurístico.

LAMA [12] Es un sistema de planificación clásico derivado de Fast Downward, por lo tanto también se basa en búsqueda heurística progresiva. Su característica principal es el uso de una pseudo-heurística derivada a partir de *Landmarks*, es decir formulas proposicionales que deben ser verdaderas en cada solución de una tarea de planificación.

1.4. En esta tesis

En esta tesis hemos trabajado sobre tres objetivos principales.

1. El problema de Split y Unsplit. Está probado que encontrar un plan puede ser muy costoso computacionalmente. Por esto, se está trabajando en algoritmos de optimización de dominios de planning. En esta tesis se utilizará el algoritmo de *Split* [13] para tratar de optimizar dichos dominios, es decir poder encontrar una solución a los problemas en un menor tiempo.

Intuitivamente, la idea del algoritmo de Split es dividir acciones que tienen una interfaz con muchos parámetros en acciones más pequeñas, para reducir el número de instancias generadas.

Split se ejecuta sobre un dominio, resultando en un dominio *Spliteado*. Esto quiere decir que si corremos un planificador sobre este nuevo dominio obtendremos un plan que no es ejecutable en el dominio original, sino que le corresponde al dominio spliteado. Para resolver este problema, durante esta tesis hemos implementado un algoritmo llamado *Unsplit*. Este algoritmo debe tomar el plan spliteado y el dominio original y devolver un plan ejecutable en el dominio original.

2. Dominios de instanciación difícil. Hemos creado dominios difíciles de instanciar, al punto de ser irresolubles en algunos casos. La idea es correr Split sobre estos dominios y comprobar en que casos los dominios spliteados pueden resolverse más rápidamente con los planificadores existentes. De esta forma al correr Unsplit sobre los planes obtenidos, obtendremos planes para los dominios originales que antes eran muy costosos o irresolubles.

3. Evaluación Empírica. En esta tesis trataremos de verificar de manera empírica si el algoritmo de Split mejora el rendimiento de los planificadores para uno o varios tipos de dominios de alto costo computacional.

En la siguiente sección se introducirá el lenguaje de definición de dominios PDDL, para luego poder presentar en la tercer sección los nuevos dominios que se implementaron en esta tesis. También se presentan los algoritmos de Split y Unsplit, necesarios para optimizar los dominios y ejecutar los tests. En la cuarta sección se presentarán los resultados obtenidos en los tests. Por último en la quinta sección se presentarán las conclusiones de todo el trabajo.

2. Split y Unsplit

En esta sección introduciremos el algoritmo de Split utilizado para optimizar dominios de planning. Comenzaremos por discutir los lenguajes utilizados para especificar dominios y problemas de planning. Para comprender mejor las diferencias entre los lenguajes de especificación de dominios daremos un ejemplo de una acción particular escrita en STRIPS.

```
(:action move
    :parameters (?from ?to)
    :precondition (and (room ?from) (room ?to) (at-robby ?from))
    :effect (and (at-robby ?to)
         (not (at-robby ?from))))
```

Como podemos ver en el ejemplo, la pre-condición de la acción es una conjunción de predicados positivos. Esta es una limitación expresiva propia de STRIPS. En algunos dominios es interesante poder escribir predicados negativos o disyunciones dentro de la pre-condición. En algunas ocasiones no podemos expresar todo lo que necesitamos solo con predicados y conjunciones. Por eso es interesante poder escribir expresiones cuantificadas. Estas se utilizan para referirse a varios objetos del dominio y no solo a objetos particulares. También es interesante poder escribir efectos condicionales, es decir que podemos escribir efectos que sólo se producen si se cumple una condición. PDDL fue diseñado para subsanar estas y otras limitaciones del poder expresivo de STRIPS. En la siguiente subsección daremos una introducción detallada a PDDL.

2.1. Introducción a PDDL

En la actualidad *Planning Domain Definition Languge* (PDDL) es el lenguaje estándar para definir dominios de planificación [14]. Es utilizado para expresar la semántica de acciones, usando pre y post-condiciones para describir la aplicabilidad y efectos de las acciones. La sintaxis está inspirada en Lisp.

Aunque el núcleo de PDDL está basado en el lenguaje utilizado en STRIPS, el lenguaje tiene mayor poder expresivo. En particular, el lenguaje incluye la capacidad de expresar una estructura de tipos para los objetos en un dominio, de ser necesario. PDDL extiende el poder expresivo del lenguaje de especificación

incluyendo la expresividad del lenguaje ADL. Esta extension incluye la capacidad de utilizar predicados negativos en la pre-condición, cuantificadores como el existencial y el universal, y efectos condicionales.

Componentes de una tarea de planning en PDDL: Una tarea de planning está compuesta por varias componentes. Los *objetos* son los objetos del mundo real que queremos representar en el sistema. Los *predicados* son las propiedades de los objetos que nos interesa representar. Estas propiedades sólo pueden ser verdaderas ó falsas. El *estado inicial* es el punto de inicio del sistema, es decir el conjunto de valores que definen al sistema antes de ejecutar cualquier acción. El *objetivo* es el estado del sistema al cual queremos llegar. Y por último las *acciones/operadores* son las herramientas que utilizamos para cambiar de estado en el sistema.

PDDL divide el problema de planificación en dos partes: *Descripción del dominio* y *Descripción del problema*. Esta división nos permite expresar diferentes problemas sobre el mismo dominio. En la *Descripción del dominio* se especifican los predicados y las acciones. En la *Descripción del problema* se especifican los objetos, el estado inicial y los objetivos.

Definición de dominios. La definición de un dominio contiene los predicados de dominio y las acciones. También puede contener tipos, constantes, hechos estáticos y muchas otras cosas, pero éstos no son compatibles con la mayoría de los planificadores.

Veamos un modelo general de un dominio escrito en PDDL. Los elementos que están entre corchetes son opcionales.

```
(define (domain domain_name)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:predicates (predicate_1_name ?a1 ?a2 ... ?an)
               (predicate_2_name ?a1 ?a2 ... ?an)
  ...)

  (:action action_1_name
    [:parameters (?p1 ?p2 ... ?pn)]
    [:precondition precond_formula]
    [:effect effect_formula]
  )

  (:action action_2_name
  ...)
)
```

Definición de acciones: Una acción toma una cantidad finita de parámetros, estos serán objetos del dominio que pueden ser de diferentes tipos. También se debe especificar una pre y post-condición, en las cuales naturalmente se utilizan

los objetos pasados por parámetro y también se puede hablar del resto de los objetos del mundo utilizando el operador existencial y el universal.

Si especificamos que el dominio es de tipo STRIPS, la pre-condición puede ser únicamente un predicado o una conjunción de predicados positivos. Si el dominio es de tipo ADL, la pre-condición puede tener además predicados negativos, conjunciones y disyunciones de predicados y formulas cuantificadas por los operadores existencial y universal. Por ejemplo:

```
(not condition_formula)
(and condition_formula1 ... condition_formula_n)
(or condition_formula1 ... condition_formula_n)
(forall (?v1 ?v2 ...) condition_formula)
(exists (?v1 ?v2 ...) condition_formula)
```

Los efectos de una acción están divididos implícitamente en *adds* y *deletes*, los deletes son los efectos negativos y están denotados por la negación. En STRIPS un efecto puede ser un predicado positivo, negativo o una conjunción de predicados. En cambio ADL permite además operadores con efecto condicional

```
(when condition_formula effect_formula)
```

Estos efectos se ejecutan sólo si se cumple la condición. Los efectos condicionales pueden colocarse dentro de cuantificadores.

```
(forall (?v1 ?v2 ...) (when condition_formula effect_formula) )
```

PDDL tiene una forma para declarar parámetros y tipos de objetos. Si se va a utilizar tipos para los parámetros, se debe declarar en los requisitos del dominio la clausula *:typing*. A continuación veamos como realizar dicha declaración.

```
(define (domain domain_name)
  (:requirements [:typing] ...)
  (:types name1 ... name_n)
  (:predicates ...)
  ...)
```

Para declarar el tipo de un parámetro de un predicado o acción se escribe:

```
?x - type_of_x
```

Una lista de parámetros del mismo tipo se puede abreviar como:

```
?x ?y ?z - type_of_xyz
```

La sintaxis es la misma para la declaración de tipos de objetos en la definición del problema.

Definición de problemas. La definición de un problema especifica los objetos presentes en la instancia del problema, la descripción del estado inicial y los objetivos.

```
(define (problem problem_name)
  (:domain domain_name)
  (:objects obj1 obj2 ... obj_n)
  (:init atom1 atom2 ... atom_n)
  (:goal condition_formula)
  )
```

El estado inicial es una lista con los átomos que son verdaderos al inicio del problema. Todos los átomos que no aparecen en esta lista se consideran automáticamente falsos. La descripción del objetivo (*goal*) es una formula con la misma forma que la pre-condición de una acción. Todos los predicados utilizados en el estado inicial y los objetivos deben estar declarados en el dominio correspondiente. A diferencia de las pre-condiciones de las acciones, en el estado inicial y los objetivos, todos los argumentos de los predicados deben ser objetos o nombres de constantes en lugar de los parámetros (*?x*) que utilizábamos en la descripción del dominio.

2.2. Split

El trabajo de Split [13] es encontrar una transformación de un dominio y un problema de planificación especificado como un dominio D y un problema P *(D,P)* en un nuevo dominio D' y problema P' *(D',P')*, tal que los tiempos de respuesta del planificador sean menores en *(D',P')* que en *(D,P)*, teniendo que existir la propiedad de que el conjunto de soluciones entre ambos problemas de planificación se preservan. Es decir, que podamos recuperar una solución de P a través de una solución de P' y viceversa.

La estrategia para la transformación consiste en tomar acciones del dominio D y dividirlas en subacciones tal que al ejecutar las subacciones en forma consecutiva se tenga los mismos efectos que al ejecutar la acción original. Al dividir una acción en subacciones, también dividimos los parámetros que toma esa acción. Esto reduce la cantidad de instanciaciones que debe realizar el planificador en cada acción, y por consiguiente debería reducirse el tiempo de respuesta del mismo. Esta estrategia de dividir las acciones de un dominio tiene como contrapartida un incremento en la cantidad de acciones resultantes haciendo que nuestros planes sean más extensos. Si pensaríamos al problema como una búsqueda en un árbol, podemos considerar que Split reduce el número de ramificaciones pero con el costo de generar una mayor profundidad en el árbol. A continuación definiremos formalmente la función de Split.

Definición 1 *Sea A un dominio. Una función Split sobre A será una función* α*: A* $\rightarrow$ *acciones***, tal que si* $\alpha(a) = a_1 ... a_k$ *se cumple que:*

I. $\forall\ l \in Lit(A).\ l \in X(a) \Leftrightarrow \exists!\ a_i\colon l \in X(a_i)$ *con* $X \in \{pre, add, del\}$
II. $\forall\ l \in Lit(A).\ l \in pre(a) \cap X(a) \Rightarrow \exists!\ a_i : l \in pre(a_i) \cap X(a_i)$ *con* $X \in \{add, del\}$

La primera condición dice que los literales de la acción original se mapean a una única subacción y todo literal de una subacción es literal de la acción original. Mientras que la segunda condición dice que si en una acción original un literal ocurre en la pre y en la post-condición, ya sea en add o del, entonces ambas ocurrencias se mapean a la misma subacción.

Veamos un ejemplo de Split de una acción.

```
(:action move
    :parameters   (?A - block ?B - block ?C - block)
    :precondition ((on A B) (clear A) (clear C))
    :effect      ((on A C) (clear B)
                (not (on A B)) (not (clear C)))
)
```

La acción *move* toma tres parámetros de tipo bloque. *(on x y)* es una relación que indica que el bloque x se encuentra sobre el bloque y. *(clear x)* denota que el bloque x no tiene ningún bloque sobre él. Suponiendo que existen m objetos de tipo bloque en el problema de planificación. El planificador deberá instanciar el esquema de acción para cada una de las posibles combinaciones de estas instancias para establecer cuales acciones son ejecutables en un estado dado. Es decir que realizará m^3 instanciaciones y verificará la ejecutabilidad de cada una de ellas.

Ahora podemos dividir la acción *move* en dos subacciones.

```
(:action move_1
    :parameters   (?A - block ?B - block)
    :precondition ((on A B) (clear A))
    :effect      ((clear B)
                (not (on A B)))
)

(:action move_2
    :parameters   (?A - block ?C - block)
    :precondition ((clear C))
    :effect      ((on A C)
                (not (clear C)))
)
```

De esta manera hemos dividido el esquema de acción *move* de tres variables en dos subacciones *move_1* y *move_2* de dos variables cada una. Por cada subacción el planificador deberá realizar m^2 instanciaciones, por lo tanto al ejecutar ambas acciones realizará un total de $2m^2$ instanciaciones, contra m^3 que realizaba con la acción original.

Esto significa una mejora en la etapa de instanciación del planificador ya que para valores grandes de m como por ejemplo $m = 100$, en el esquema original tenemos 1.000.000 de instancias contra 20.000 instancias en el esquema spliteado.

Es importante tener en cuenta que en el nuevo esquema de acciones, estas no pueden ser ejecutadas en cualquier orden y además deben ser ejecutadas de forma atómica, es decir que no puede ejecutarse ninguna otra acción entre medio de las subacciones. También es necesario notar que no es trivial realizar un split, ya que no todas las particiones son posibles de ejecutar, ni tampoco todas las particiones preservan el objetivo original. Por ejemplo, consideremos la siguiente partición.

```
(:action move_1
    :parameters   (?A - block ?B - block)
    :precondition ((on A B))
    :effect      ((clear B)
                 (not (on A B)))
)

(:action move_2
    :parameters   (?A - block ?C - block)
    :precondition ((clear A) (clear C))
    :effect      ((on A C)
                 (not (clear C)))
)
```

Este split no es ejecutable ya que *move_1* permite que exista un bloque encima de A, mientras que *move_2* lo prohíbe en su pre-condición.

Para que un split sea válido debemos asegurar lo siguiente:

1. Las subacciones son instanciadas consistentemente, es decir que los parámetros compartidos son asignados al mismo objeto.
2. Las subacciones de la misma acción son ejecutadas en bloque, es decir que no se ejecuta ninguna subacción de otra acción entre medio.
3. Si un átomo ocurre en la precondición de una acción, este debe ocurrir en la precondición en el split. Si un delete ocurre en una acción este debe ocurrir antes que los adds en el split.

En un dominio pueden existir muchos splits posibles, pero no todos son útiles. Por lo tanto el algoritmo de Split toma un parámetro *gamma*, el cual permite regular cuanto se desea splitear al dominio. Este parámetro es un valor numérico entre 0 y 1. Cuanto más se aproxima a 0, más se splitea, mientras que cuando gamma vale 1, no se divide ninguna acción.

2.3. Unsplit

Unsplit es un algoritmo creado para generar un plan con las acciones originales de un dominio a partir de un plan *spliteado* de ese mismo dominio. Esto

nos permite resolver un problema spliteado con un planificador y recuperar un plan válido para el dominio original.

Por ejemplo, si tomamos la acción *move* presentada en la sección anterior y un split válido para esa acción. Un plan spliteado podría ser el siguiente:

```
MOVE_1 A B
MOVE_2 A C
```

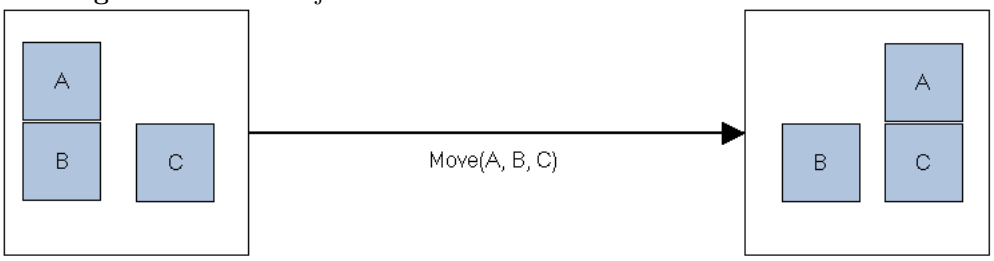
Pero este no es un plan válido sobre el dominio original, ya que el original solo tiene la acción *move*. Unsplit debe lograr reconstruir la acción *move*, pero para hacer esto necesita información adicional a la que podemos encontrar en el plan spliteado. La información faltante es a que parámetro de la acción original corresponde cada parámetro de las acciones spliteadas. Ya que la acción *move_1* toma dos parámetros de la acción *move*, pero no sabemos cual de los tres parámetros es cada uno. Para resolver este problema se tuvo que modificar el algoritmo de Split para que retorne esa información en un archivo *log*. Ese archivo contiene la siguiente información:

```
move 1 - 1 2
move 2 - 1 3
```

Este archivo nos indica por ejemplo que la acción *move_2* toma el primer parámetro de la acción original como su primer parámetro y el tercer parámetro de la acción original como su segundo parámetro. Ahora Unsplit esta en condiciones de reconstruir un plan con las acciones originales. El resultado de correr Unsplit sobre este plan spliteado con este log, es el siguiente:

```
MOVE A B C
```

Figura2. Efecto de ejecutar la acción Move en el dominio Blocks.



Como utilizar Unsplit: Unsplit es un script escrito en Python, y se debe correr con el siguiente comando:

```
python unsplit.py [-p <plan>] [-l <log>] [-o <output>]
```

Todos los argumentos que toma son opcionales y cada una representa lo siguiente:
<*plan*>: Dirección del plan spliteado.
<*log*>: Dirección del log generado cuando se spliteo el dominio original
<*output*>: Dirección del archivo donde se guardara el plan despliteado
En el caso en que no se ingrese alguna de las opciones mostradas anteriormente, se tomaran los siguientes valores por defecto:
<*plan*>: "unsplit/plan"
<*log*>: "unsplit/log"
<*output*>: "unsplit/out"
Siempre considerando "unsplit/" como la dirección de la carpeta donde se encuentra el archivo *unsplit.py*

En cuanto a las decisiones de diseño, se implementó todo el script en una sola función ya que el código no es complejo de leer ni de entender. El código está abundantemente comentado y además, para facilitar la lectura, se sigue el estándar de coding style PEP8 [15]. El mismo fue verificado con Flake8.

Se buscó utilizar al máximo las funciones que provee Python para realizar todas las acciones en la menor cantidad de líneas posibles.

3. Diseño de nuevos dominios

Como comentamos anteriormente, parte de los objetivos de esta tesis fue el diseño y testeo de dominios de planning que pudieran resultar interesantes para las funciones de Split y Unsplit. En esta sección describiremos los dominios implementados.

3.1. Crossing

Descripción: En este dominio se representa al clásico problema del granjero que intenta cruzar el río acompañado con un lobo, una gallina y una bolsa de maíz. El problema radica en que para cruzar el río solo se dispone de una balsa tan pequeña que solo entran dos objetos, por ejemplo el granjero y la gallina. También tenemos la restricción de que el granjero debe ir en la balsa en cada cruce para poder navegarla, y por último debe cuidar de que los objetos no se coman entre si, ya que si el lobo queda solo con la gallina se la comería, y la gallina se comería al maíz. El objetivo final es llegar a la otra costa del río con todos los objetos intactos.

Dominio: Para formalizar este dominio necesitamos utilizar ADL y tipos de objetos

```
(define (domain crossing)
    (:requirements :adl :typing)
    (:types object coast)
```

```
    (:predicates (on ?x - object ?y - coast)
                 (can_eat ?x - object ?y - object)
                 (is_farmer ?x - object)
    )
   ...
```

Definimos el predicado *on*, el cual indica en que costa esta cada objeto, el predicado *can_eat* que determina que objeto X puede comer a otro objeto Y, y por último el predicado *is_farmer* que indica que objeto es el granjero.

Acciones: Definimos dos acciones que nos permitirán lograr el objetivo:

```
(:action cross_1
:parameters   (?farmer - object ?from - coast ?to - coast)
:precondition (and
                  (on ?farmer ?from)
                  (is_farmer ?farmer)
                  (not (= ?from ?to))
                  (not
                      (exists (?ob1 - object ?ob2 - object)
                          (and
                              (not (= ?ob1 ?farmer))
                              (not (= ?ob2 ?farmer))
                              (on ?ob1 ?from) (on ?ob2 ?from)
                              (can_eat ?ob1 ?ob2)))))
:effect (and
           (on ?farmer ?to) (not (on ?farmer ?from))
        )
)
```

Como podemos ver esta acción toma como parámetros al granjero, a la costa de salida y a la costa de llegada. En la pre-condición pedimos que el primer objeto de los parámetros sea efectivamente el granjero, que esté en la costa de salida, es decir donde esta la balsa. Y por último pedimos que no exista ningún objeto en la costa de salida que pueda comer a otro objeto.

En la post-condición indicamos que al aplicar esta acción el granjero deja de estar en la costa de salida y ahora se encuentra en la costa de llegada (implícitamente junto con la balsa).

En definitiva esta acción le permite al granjero cruzar el río solo. También definimos la acción:

```
(:action cross_2
:parameters   (?farmer - object ?ob - object ?from - coast ?to -
    coast)
:precondition (and
                  (on ?farmer ?from)
                  (on ?ob ?from)
```

```
                    (is_farmer ?farmer)
                    (not (is_farmer ?ob))
                    (not (= ?from ?to))
                    (not
                        (exists (?ob1 - object ?ob2 - object)
                            (and
                                (not (= ?ob1 ?farmer))
                                (not (= ?ob2 ?farmer))
                                (not (= ?ob1 ?ob))
                                (not (= ?ob2 ?ob))
                                (on ?ob1 ?from) (on ?ob2 ?from)
                                (can_eat ?ob1 ?ob2)))))
:effect (and
           (on ?farmer ?to) (not (on ?farmer ?from))
           (on ?ob ?to) (not (on ?ob ?from))
         )
)
```

Esta acción toma como parámetros al granjero y a un objeto acompañante además de las costas de salida y de llegada.

La pre-condición es muy similar a la de la acción *cross_1* solo que al tomar un parámetro mas debemos pedir que este segundo objeto no sea el granjero y que tanto el granjero como el acompañante estén en la costa de salida. Y por último pedimos que no exista ningún objeto en la costa de salida que pueda comer a otro objeto.

En la post-condición indicamos que al aplicar esta acción el granjero y el acompañante dejan de estar en la costa de salida y ahora se encuentran en la costa de llegada.

Esta acción le permiten al granjero poder cruzar el río acompañado, y con la acción anterior este podría volver solo a buscar otro objeto, si lo necesitara.

Problema: A continuación mostraremos el problema descripto anteriormente con todas sus restricciones en lenguaje ADL.

```
(define (problem crossing)
  (:domain crossing)
  (:objects near far - coast
            farmer fox chicken corn - object)
  (:init
   (is_farmer farmer)
   (on farmer near)
   (on fox near)
   (on chicken near)
   (on corn near)
   (can_eat fox chicken)
   (can_eat chicken corn)
```

```
  )
  (:goal
    (and (on farmer far) (on fox far) (on chicken far)
         (on corn far))
  )
)
```

Al correr un planificador sobre este dominio y este problema obtenemos el siguiente plan.

```
step   0: cross_2 farmer chicken near far
       1: cross_1 farmer far near
       2: cross_2 farmer fox near far
       3: cross_2 farmer chicken far near
       4: cross_2 farmer corn near far
       5: cross_1 farmer far near
       6: cross_2 farmer chicken near far
```

Básicamente es una lista de pasos donde en cada línea tenemos, el nombre de la acción a ejecutar con todos sus parámetros instanciados. Por ejemplo el primer paso es ejecutar la acción *cross_2*, es decir cruzar el río el granjero con un acompañante, en este caso es con el pollo, desde la costa cercana hasta la costa lejana.

Comentarios: Este dominio no debería resultar problemático para un planificador. Puede ser interesante investigar el efecto de agregar más objetos. Ya que las acciones de este dominio tienen interfaces de 3 y 4 parámetros, y no hay predicados que los relacionen a todos. Este dominio es un potencial candidato para optimizar con Split en problemas con un número elevado de objetos. El objetivo principal de este dominio fue definir una codificación que pudiera generalizarse al dominio descripto a continuación.

3.2. River Game

Descripción: Este dominio es una generalización del problema de *Crossing*, se escribió pensando en poder escalarlo hasta donde sea necesario para realizar pruebas de rendimiento.

El problema básico consiste en que una familia constituida por un padre, una madre y sus hijos, tanto mujeres como varones pueden cruzar el río en una balsa que soporta solo dos personas al mismo tiempo. También debe poder cruzar el río un policía y un ladrón. Las dificultades con las que nos encontramos son que el padre no puede dejar a los hijos varones con la madre porque esta los golpea. La madre no puede dejar a las hijas mujeres con el padre por la misma razón, el policía puede estar y cruzar el río con cualquier persona, pero no puede dejar al ladrón con nadie ya que este los golpea a todos. La balsa debe ser dirigida por algún adulto, es decir que la debe conducir el padre, la madre o el policía y estos pueden estar acompañados por cualquiera que no viole las restricciones

antes mencionadas. Por ejemplo el padre puede cruzar solo o con un hijo varón pero no con una hija mujer o con el ladrón. El objetivo es que todos puedan llegar a la otra costa sin ser golpeados.

Dominio: En este dominio, al igual que en el anterior necesitamos ADL y tipos de objetos. Los tipos declarados son: objeto y costa.

```
(define (domain rivergame)
    (:requirements :adl :typing)
    (:types object coast)
    (:predicates (on ?x - object ?y - coast)
                 (can_punch ?x - object ?y - object ?z - object)
                 ;'x' can punch 'y' without 'z'
                 (is_dad ?x - object)
                 (is_mom ?x - object)
                 (is_boy ?x - object)
                 (is_girl ?x - object)
                 (is_police ?x - object)
                 (is_thief ?x - object)
                 (boat_on ?x - coast)
    )
    ...
```

Los predicados que utilizamos son: *on*, que indica en que costa esta cada objeto, *can_punch* que indica que un objeto X puede golpear a un objeto Y sólo si no se encuentra presente el objeto Z.

También utilizamos los predicados *is*, por ejemplo *is_dad* para indicar que ese objeto es un padre, y por último el predicado *boat_on* el cual indica en que costa se encuentra la balsa. Este último predicado no era necesario en el problema de *Crossing* porque el único que podía navegar la balsa era el granjero. Y por lo tanto podríamos asumir que la balsa estaba en la misma costa donde se encontraba el granjero.

Acciones: Para lograr los objetivos que buscamos necesitamos definir dos acciones, una para que un adulto cruce el río solo y otra para que lo cruce acompañado con una persona.

```
(:action cross_1
:parameters   (?adult - object ?from - coast ?to - coast)
:precondition (and
                  (on ?adult ?from)
                  (or (is_dad ?adult) (is_mom ?adult) (is_police
                      ?adult))
                  (not (= ?from ?to))
                  (boat_on ?from)
                  (not
                      (exists (?ob1 - object ?ob2 - object)
```

```
                        (and
                            (not (= ?ob1 ?adult))
                            (not (= ?ob2 ?adult))
                            (on ?ob1 ?from) (on ?ob2 ?from)
                            (can_punch ?ob1 ?ob2 ?adult)
                        )
                    )
                )
                (not
                    (exists (?ob1 - object ?ob2 - object)
                        (and
                            (not (= ?ob1 ?adult))
                            (not (= ?ob2 ?adult))
                            (on ?ob1 ?from)
                            (on ?ob2 ?to)
                            (can_punch ?adult ?ob2 ?ob1)))))
:effect (and
           (on ?adult ?to) (not (on ?adult ?from))
           (boat_on ?to) (not (boat_on ?from))
         )
)
```

En la pre-condición se pide que el primer objeto de los parámetros sea un adulto, es decir que sea el padre, la madre o el policía, y además que se encuentre en la costa de salida junto con la balsa.

Luego se pide que no exista ningún objeto en la costa de salida que pueda golpear a otro sin el adulto que navega la balsa. Y por último se necesita que el adulto que navega la balsa no pueda golpear a nadie que se encuentre en la costa de llegada sin la presencia de algún objeto que se encuentre en la costa de salida.

En la post-condición indicamos que al aplicar esta acción el adulto que navega y la balsa dejan de estar en la costa de salida y ahora se encuentran en la costa de llegada.

```
(:action cross_2
:parameters    (?adult - object ?child - object ?from - coast ?to
    - coast)
:precondition  (and
                    (on ?adult ?from)
                    (on ?child ?from)
                    (or (is_dad ?adult) (is_mom ?adult) (is_police
                        ?adult))
                    (not (= ?adult ?child))
                    (not (= ?from ?to))
                    (boat_on ?from)
                    (not
```

```
                    (exists (?ob1 - object ?ob2 - object)
                        (and
                            (not (= ?ob1 ?adult))
                            (not (= ?ob2 ?adult))
                            (not (= ?ob1 ?child))
                            (not (= ?ob2 ?child))
                            (on ?ob1 ?from) (on ?ob2 ?from)
                            (or
                                (can_punch ?ob1 ?ob2 ?adult)
                                (can_punch ?ob1 ?ob2 ?child)))))
                (not
                    (exists (?ob1 - object ?ob2 - object)
                        (and
                            (not (= ?ob1 ?adult))
                            (not (= ?ob2 ?adult))
                            (not (= ?ob1 ?child))
                            (not (= ?ob2 ?child))
                            (on ?ob1 ?from)
                            (on ?ob2 ?to)
                            (or
                                (can_punch ?adult ?ob2 ?ob1)
                                (can_punch ?child ?ob2 ?ob1)))))
                (not
                    (exists (?ob1 - object)
                        (can_punch ?adult ?child ?ob1)
                    )
                )
            )
:effect (and
            (on ?adult ?to) (not (on ?adult ?from))
            (on ?child ?to) (not (on ?child ?from))
            (boat_on ?to) (not (boat_on ?from))
        )
)
```

Similarmente a la acción anterior se pide que un objeto sea un adulto y que el otro sea cualquier otro excepto el mismo, ambos deben estar en la costa de salida junto con la balsa.

Ahora se pide que no exista ningún par de objetos en la costa de salida que puedan golpearse con la ausencia de alguna persona de la balsa.

Luego se pide que ninguna persona de la balsa pueda golpear a alguna persona de la costa de llegada con la ausencia de algún objeto en la costa de salida.

Y por último se pide que el adulto que navega la balsa no pueda golpear al acompañante con la ausencia de cualquier persona.

Problema: A continuación se muestra la formalización del problema básico descripto anteriormente en lenguaje ADL.

```
(define (problem rivergame)
  (:domain rivergame)
  (:objects near far - coast
            dad mom police thief boy1 girl1 boy2 girl2 - object)
  (:init
   (is_dad dad)
   (is_mom mom)
   (is_police police)
   (is_thief thief)
   (is_boy boy1) (is_boy boy2)
   (is_girl girl1) (is_girl girl2)
   (on dad near)
   (on mom near)
   (on police near)
   (on thief near)
   (on boy1 near)
   (on boy2 near)
   (on girl1 near)
   (on girl2 near)
   (can_punch dad girl1 mom)
   (can_punch dad girl2 mom)
   (can_punch mom boy1 dad)
   (can_punch mom boy2 dad)
   (can_punch thief dad police)
   (can_punch thief mom police)
   (can_punch thief boy1 police)
   (can_punch thief boy2 police)
   (can_punch thief girl1 police)
   (can_punch thief girl2 police)
   (boat_on near)
  )
  (:goal
    (and (on dad far) (on mom far) (on police far) (on thief far)
         (on boy1 far) (on girl1 far) (on boy2 far) (on girl2 far))
  )
)
```

Al correr un planificador sobre este dominio y este problema obtenemos el siguiente plan.

```
step   0: cross_2 police thief near far
       1: cross_1 police far near
       2: cross_2 police boy2 near far
```

```
 3: cross_2 police thief far near
 4: cross_2 dad boy1 near far
 5: cross_1 dad far near
 6: cross_2 mom dad near far
 7: cross_1 mom far near
 8: cross_2 police thief near far
 9: cross_1 dad far near
10: cross_2 mom dad near far
11: cross_1 mom far near
12: cross_2 mom girl1 near far
13: cross_2 police thief far near
14: cross_2 police girl2 near far
15: cross_1 police far near
16: cross_2 police thief near far
```

Similarmente al problema de *crossing*, la acción *cross_1* sirve para que un adulto solo cruce el río, mientras que *cross_2* sirve para cruzar acompañado con otra persona.

Comentarios: Este problema se intentará escalar sobre la cantidad de hijos varones y mujeres y sobre la cantidad de pares de policías y ladrones. Ver la cuarta sección sobre testing para una discusión de los resultados obtenidos.

3.3. Checkers

Descripción: En este dominio se describe el juego de las damas chinas simplificado en varios aspectos. Básicamente el objetivo del juego es que todas las fichas del jugador lleguen a cierta posición definida en el problema particular. En esta formalización solo hay un jugador que puede mover sus fichas hacia un casillero que se encuentre libre, es decir que no tenga ninguna ficha sobre el. Pero además para poder llegar hasta ese casillero la ficha debe *saltar* a una ficha que se encuentre en el casillero contiguo al que se desea llegar, considerando la dirección con la que se realiza el salto.

Solo se puede saltar en ocho direcciones, izquierda, derecha, arriba, abajo y todas las diagonales que se forman entre estas cuatro direcciones principales. Es decir que si un jugador desea mover una ficha hacia arriba, en el casillero que se encuentra inmediatamente arriba de esa ficha debe haber una ficha, y el casillero que se encuentra dos posiciones hacia arriba debe estar libre. En este caso el jugador puede ejecutar su movimiento y su ficha queda finalmente en el casillero antes libre.

El jugador puede saltar cuantas veces quiera, siempre y cuando se cumplan las condiciones de salto.

```
(define (domain checkers)
    (:requirements :adl :typing)
    (:types block chip way)
```

```
(:predicates (on ?x - chip ?y - block)
             (connected ?x - block ?y - block ?z - way)
)
```

En este dominio se requiere ADL y tipos de objetos. Los tipos declarados son: *block* el cual representa a los casilleros donde se encuentran las fichas, *chip* el cual representa las fichas, y por último *way* el cual agrupa las ocho direcciones posibles.

En cuanto a los predicados solo definimos dos, el predicado *on* el cual indica en que bloque esta parada una ficha. Y el predicado *connected* el cual indica que un bloque X esta conectado a otro bloque Y en la dirección Z, mirando desde el punto de vista de X, por ejemplo:

```
(on block_1 block_2 right)
```

indica que block_2 esta a la derecha de block_1, pero block_1 no se encuentra a la izquierda de block_2 todavía, para que la conexión este completa debemos declarar:

```
(on block_2 block_1 left)
```

De esta forma podemos construir un tablero N por M tan grande como lo necesitemos.

Acciones: En este dominio solo necesitamos definir una acción.

```
(:action move
:parameters   (?chip - chip ?from - block ?way - way ?middle -
    block ?to - block)
:precondition (and
                  (on ?chip ?from)
                  (connected ?from ?middle ?way)
                  (exists (?chip_2 - chip) (on ?chip_2 ?middle))
                  (connected ?middle ?to ?way)
                  (forall (?chip_3 - chip) (not (on ?chip_3 ?to)))
              )
:effect (and
           (on ?chip ?to) (not (on ?chip ?from))
        )
)
```

En la pre-condición de esta acción se pide que la ficha se encuentre en el casillero de salida. También que el casillero de salida esté conectado con un casillero intermedio, el cual debe tener alguna ficha sobre el y además estar conectado con el casillero de llegada. Por último se pide que el casillero de llegada se encuentre libre. En la post-condición se indica que la ficha que se encontraba en el casillero de salida ahora se encuentra en el casillero de llegada.

Problema: A continuación se muestra la formalización del problema 3 x 3 con 2 fichas en lenguaje ADL.

```
(define (problem checkers_3_3)
  (:domain checkers)
  (:objects chip_1 chip_2 - chip
    block_0_0 block_0_1 block_0_2 block_1_0 block_1_1 block_1_2
        block_2_0 block_2_1 block_2_2 - block
    up down left right up_right up_left down_right down_left -
        way)
  (:init (on chip_1 block_0_0)
    (on chip_2 block_1_1)
    (connected block_0_0 block_0_1 up)
    (connected block_0_0 block_1_0 right)
    (connected block_0_0 block_1_1 up_right)
    (connected block_0_1 block_0_0 down)
    (connected block_0_1 block_0_2 up)
    (connected block_0_1 block_1_0 down_right)
    (connected block_0_1 block_1_1 right)
    (connected block_0_1 block_1_2 up_right)
    (connected block_0_2 block_0_1 down)
    (connected block_0_2 block_1_1 down_right)
    (connected block_0_2 block_1_2 right)
    (connected block_1_0 block_0_0 left)
    (connected block_1_0 block_0_1 up_left)
    (connected block_1_0 block_1_1 up)
    (connected block_1_0 block_2_0 right)
    (connected block_1_0 block_2_1 up_right)
    (connected block_1_1 block_0_0 down_left)
    (connected block_1_1 block_0_1 left)
    (connected block_1_1 block_0_2 up_left)
    (connected block_1_1 block_1_0 down)
    (connected block_1_1 block_1_2 up)
    (connected block_1_1 block_2_0 down_right)
    (connected block_1_1 block_2_1 right)
    (connected block_1_1 block_2_2 up_right)
    (connected block_1_2 block_0_1 down_left)
    (connected block_1_2 block_0_2 left)
    (connected block_1_2 block_1_1 down)
    (connected block_1_2 block_2_1 down_right)
    (connected block_1_2 block_2_2 right)
    (connected block_2_0 block_1_0 left)
    (connected block_2_0 block_1_1 up_left)
    (connected block_2_0 block_2_1 up)
    (connected block_2_1 block_1_0 down_left)
    (connected block_2_1 block_1_1 left)
```

```
    (connected block_2_1 block_1_2 up_left)
    (connected block_2_1 block_2_0 down)
    (connected block_2_1 block_2_2 up)
    (connected block_2_2 block_1_1 down_left)
    (connected block_2_2 block_1_2 left)
    (connected block_2_2 block_2_1 down))
  (:goal
    (and (on chip_1 block_2_2))
  )
)
```

Este problema representa un tablero con tres filas, tres columnas y dos fichas. Los nueve bloques están conectados entre si formando un cuadrado. La ficha número uno se encuentra en el bloque 0 0, es decir el bloque de la primera fila y la primera columna. Mientras que la ficha número dos se encuentra en el bloque 1 1, es decir en el centro del tablero. El objetivo es que la ficha número uno llegue al bloque 2 2. Por lo tanto es fácil ver que la ficha se encuentra a un salto de la meta y cumple con todas las condiciones de salto. Al correr un planificador sobre este dominio y este problema obtenemos el siguiente plan.

```
step   0: move chip_1 block_0_0 up_right block_1_1 block_2_2
```

Comentarios: Se escribió este dominio para probar la optimización que realiza el algoritmo de Split para esta clase de problemas que generan muchas instancias. Junto con este dominio se implementó un generador de problemas, es decir un algoritmo que dadas las dimensiones del tablero, la cantidad de fichas y opcionalmente la posición de cada ficha nos genera el código PDDL del problema indicado automáticamente. En la siguiente sección se probarán problemas más complejos sobre este dominio.

3.4. Cards

Descripción: Este dominio representa una baraja de cartas numéricas, donde cada carta se encuentra inicialmente en una posición especifica dentro de la baraja con respecto a las otras cartas. Y el objetivo es poder ordenar las cartas de menor a mayor.

```
(define (domain cards)
    (:requirements :adl :typing)
    (:types card position)
    (:predicates (on ?x - card ?y - position)
                 (smaller ?x - card ?y - card))
   ...
)
```

En este dominio ser requiere ADL y tipos de objetos. Los tipos declarados son: *card*, el cual representa las cartas de la baraja. Y por último *position*, el cual representa las posiciones de las cartas dentro de la baraja.

Contamos con dos predicados. El predicado *on*, el cual indica la posición actual de una carta. Y el predicado *smaller* el cual indica que la primera carta tiene una numeración mas baja que la segunda carta.

Acciones: En este dominio solo necesitamos definir una acción.

```
(:action swap
   :parameters   (?card1 - card ?card2 - card ?from - position
      ?to - position)
    :precondition (and
                      (on ?card1 ?from) (not (on ?card1 ?to))
                      (on ?card2 ?to) (not (on ?card2 ?from))
                  )
    :effect (and
               (on ?card2 ?from) (not (on ?card2 ?to))
               (on ?card1 ?to) (not (on ?card1 ?from))
            )
)
```

Esta acción sirve para intercambiar la posición de una carta con otra. En la pre-condición se pide básicamente que ambas cartas se encuentren en posiciones distintas, para que el intercambio tenga sentido. En la post-condición se indica que la carta número uno queda ubicada en la posición donde se encontraba la carta número dos y viceversa.

Problema: A continuación se muestra la formalización del problema con 3 cartas en lenguaje ADL.

```
(define (problem cards)
  (:domain cards)
  (:objects card_1 card_2 card_3 - card pos_1 pos_2 pos_3 -
     position)
  (:init (smaller card_1 card_2)
   (smaller card_1 card_3)
    (smaller card_2 card_3)
    (on card_1 pos_1)
    (on card_2 pos_3)
    (on card_3 pos_2))
  (:goal (and (on card_1 pos_1)
    (on card_2 pos_2)
    (on card_3 pos_3)))
)
```

Este problema cuenta con tres cartas, donde la carta número uno es menor que la número dos y la número tres, y la carta número dos es menor que la número tres. Inicialmente la carta número uno esta en la posición que le corresponde, pero la número dos se encuentra en la posición tres y la número tres en la posición dos. La meta del problema es lograr ordenar las cartas de menor a mayor. Por lo tanto un plan válido para este problema sería intercambiar la carta número dos con la carta número tres.

Comentarios: Este es un dominio muy simple de entender y escribir pero lleva consigo el problema de ordenamiento, este problema es conocido por tener una gran complejidad algorítmica. Por lo tanto al incrementar el número de cartas desordenadas dentro del problema se puede conseguir un tiempo de búsqueda tan alto como se desee. Esto es interesante para ver como se comporta la técnica de Split ante esta clase de problemas.

4. Tests

Se realizó una evaluación empírica sobre varios dominios introducidos en la sección anterior y uno seleccionado de la IPC 2011.

El objetivo de esta evaluación es comprobar si el algoritmo de Split mejora el rendimiento de los planificadores para encontrar algún plan válido sobre estos dominios.

Consideraremos una mejora de rendimiento el obtener un plan de mayor calidad, es decir un plan de menor costo, y obtener un plan en un tiempo menor.

Para cada dominio se corrió el planificador FF, para obtener el plan que llamaremos *original*. Luego se corrió el algoritmo de Split sobre el domino y un problema particular para obtener el dominio spliteado y el log. Con el dominio spliteado se corrió el planner . En los casos en los que se obtuvo un resultado, con el plan obtenido y el log se corrió el algoritmo de Unsplit y se recuperó un plan con las acciones del dominio original al cual llamaremos plan *unspliteado*.

Para obtener mejores resultados se corrió el algoritmo de Split con tres parámetros *gamma* distintos.

Basados en estos criterios estudiaremos los resultados de cada dominio seleccionado.

Como leer los resultados. A continuación mostraremos una tabla de ejemplo para explicar el significado de cada columna.

Cada fila representa los resultados obtenidos para cada Split realizado, excepto la primera fila, que representa al dominio original sin aplicar el algoritmo de Split.

La columna “Split” indica en cuantas acciones se dividió el dominio al splitearlo. “Max I” hace referencia a la cantidad de parámetros que toma la acción con la interfaz más grande del dominio. “Max O” se refiere a la cantidad de objetos que posee el tipo que más objetos tiene en el dominio. “Costo” indica el costo unitario de cada plan despliteado, es decir la cantidad de acciones que

Problema									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	0 acc.	0	0	0	0	0 + 0	0	0	0
0.8	0 acc.	0	0	0	0	0 + 0	0	0	0
0.5	0 acc.	0	0	0	0	0 + 0	0	0	0
0.3	0 acc.	0	0	0	0	0 + 0	0	0	0

Cuadro1. Muestra.

tiene el plan despliteado. "Estados" indica la cantidad de estados evaluados durante la búsqueda. "Ins" representa la cantidad de instancias fáciles y difíciles, esta separación es propia del planificador FF, y la representamos como la suma de las instancias fáciles y las instancias difíciles para el planificador. Las últimas tres columnas son los tiempos de búsqueda, instanciación y total, representados en segundos.

4.1. Test Gripper

Resultados obtenidos.

Problema con 4 pelotas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	3 acc.	3	4	11	16	36 + 0	0	0	0
0.8	3 acc.	3	4	11	16	36 + 0	0	0	0
0.5	5 acc.	3	4	11	67	36 + 0	0	0	0
0.3	7 acc.	2	4	13	228	40 + 0	0	0	0
Problema con 20 pelotas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	3 acc.	3	20	59	80	164 + 0	0	0	0
0.8	3 acc.	3	20	59	80	164 + 0	0	0	0
0.5	5 acc.	3	20	59	1099	164 + 0	0.02	0	0.02
0.3	7 acc.	2	20	77	13868	168 + 0	0.19	0	0.19

Cuadro2. Gripper.

Podemos ver que en el problema con 4 pelotas no se aprecian grandes diferencias entre el dominio original y los dominios spliteados. Esto se debe a que es un problema muy pequeño para apreciar diferencias. Solo podemos ver que con gamma 0.3 se incremento considerablemente la cantidad de estados evaluados.

En el problema con 20 pelotas se aprecia mejor que, cuanto más se divide las acciones, más se incrementa la cantidad de estados evaluados. En el caso del split con gamma 0.5, podemos ver que el costo del plan y el tiempo son muy buenos

considerando que el plan obtenido en el caso original es el óptimo, por lo cual no podríamos optimizarlo más pero si igualarlo. En el caso del split con gamma 0.3 se reduce la máxima interfaz de las acciones, lo cual debería reducir el número de instancias, pero al haber aumentado tanto el número de acciones esto no sucede. También aumento considerablemente el costo del plan y el tiempo de búsqueda.

4.2. Test Cards

Resultados obtenidos.

Problema con 10 cartas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	1 acc.	4	10	8	9	40000 + 0	0.01	0	0.03
0.8	1 acc.	4	10	8	9	40000 + 0	0.01	0	0.03
0.5	1 acc.	4	10	8	9	40000 + 0	0.01	0	0.03
0.3	7 acc.	2	10	8	13200	1400 + 0	0.29	0	0.29
Problema con 30 cartas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	1 acc.	4	30	27	28	3240000 + 0	2.92	0.43	5.26
0.8	1 acc.	4	30	27	28	3240000 + 0	2.92	0.43	5.26
0.5	1 acc.	4	30	27	28	3240000 + 0	2.86	0.42	5.05
0.3	7 acc.	2	30	27	892729	12600 + 0	1168.02	0	1168.05

Cuadro3. Cards.

En ambos problemas podemos ver a mayor o menor escala las mismas características. Los splits con gamma 0.8 y 0.5 no generaron nuevas acciones, por lo tanto los resultados obtenidos son idénticos al original. También es importante notar que el único Split encontrado tiene siete acciones, mientras que el original solo una, esto explica la gran diferencia entre los valores obtenidos.

Analizando el split con gamma 0.3 vemos nuevamente que al incrementar el número de acciones, se incrementa el número de estados, pero en este caso el incremento es mucho más notable que en Gripper. También notemos que en el dominio original se genera una gran cantidad de instancias, y en el dominio spliteado con gamma 0.3 se reduce drásticamente ese número. Esto provoca que en el problema con 30 cartas se reduzca el tiempo de instanciación de 0.43 segundos a 0. La diferencia de tiempo no es tan notable porque en este dominio no se tiene problemas de instanciación, es decir que a pesar de tener una gran cantidad de instancias estas son fáciles de instanciar para el planificador. Mientras que el gran incremento en los estados sí provoca un impacto importante en el tiempo de búsqueda pasando de menos de 3 segundos a casi 20 minutos.

4.3. Test River Game

Este dominio es la generalización de Crossing, por lo tanto se creo para poder incrementar el número de objetos con un generador de problemas. Al incrementar solo la cantidad de hijos el problema no tiene solución, lo mismo sucede al incrementar la cantidad de ladrones.

Por lo tanto se hicieron pruebas incrementando las parejas de policías y ladrones, padres e hijos, y madres e hijas. Estos nuevos problemas si tienen solución, pero no representan dificultades para el planificador. Por lo tanto no son dominios interesantes para optimizar.

Y además no fue posible encontrar algún Split, por lo tanto no hay resultados para reportar.

4.4. Test Checkers

Resultados obtenidos.

Problema 7 x 6 con 7 fichas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	1 acc.	5	42	6	9	0 + 9604	0.01	8.30	8.33
0.8	1 acc.	5	42	6	8	0 + 9604	0.00	2.20	2.21
0.5	3 acc.	3	42	6	507	12610 + 1834	0.70	0.03	0.76
0.3	-	-	-	-	-	-	-	-	-
Problema 7 x 7 con 7 fichas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	1 acc.	5	49	-	-	-	-	-	Time out
0.8	1 acc.	5	49	-	-	-	-	-	Time out
0.5	3 acc.	3	49	7	321	17119 + 2184	0.87	0.08	1.00
0.3	-	-	-	-	-	-	-	-	-
Problema 8 x 8 con 8 fichas									
gamma	Split	Max I	Max O	Costo	Estados	Ins	TB	TI	TT
original	1 acc.	5	64	-	-	-	-	-	Time out
0.8	1 acc.	5	64	-	-	-	-	-	Time out
0.5	3 acc.	3	64	6	1221	33188 + 3360	8.19	0.12	8.43
0.3	-	-	-	-	-	-	-	-	-

Cuadro4. Checkers.

En este dominio el Split con gamma 0.8 no genero nuevas acciones por lo tanto los valores son muy similares al original. También el Split con gamma 0.3 genero las mismas acciones que el Split con gamma 0.5, por lo tanto no lo consideraremos.

En el problema 7 x 6 con 7 fichas podemos ver las mismas características que analizamos en los dominios anteriores, es decir que Split al incrementar la cantidad de acciones reduce la máxima interfaz. Esto provoca una gran reducción en el número de instancias. Particularmente en este dominio, al tener una gran cantidad de instancias difíciles para el planificador, el tiempo de instanciación es alto. Split reduce considerablemente el número de instancias difíciles, pero a cambio genera muchas instancias fáciles.

Podemos ver que a pesar de esto el tiempo de instanciación se ve reducido considerablemente. Pero al aumentar el número de acciones también incrementamos la cantidad de estados, aun que en este dominio no se incrementa en una cantidad significativa, por lo tanto el tiempo de búsqueda no incrementa mucho.

Cuando observamos el tiempo total notamos que la ganancia en el tiempo de instanciación compensa el tiempo de búsqueda perdido y se logra reducir el tiempo total significativamente.

Cuando lo testeamos a mayor escala con los problemas 7 x 7 con 7 fichas y 8 x 8 con 8 fichas. Podemos ver que las características antes mencionadas se mantienen. Con la diferencia que en el dominio original el planificador no logra encontrar un plan ya que excede el tiempo limite establecido. Pero en el dominio spliteado logra terminar en un tiempo muy bueno y a pesar de la gran cantidad de instancias, el tiempo de instanciación se mantiene debajo de un segundo.

5. Conclusión

5.1. Sobre el trabajo concreto

En todos los dominios analizados pudimos observar características muy similares a pesar de ser dominios muy diferentes entre ellos. A continuación analizaremos los resultados obtenidos en cada dominio para poder dar las conclusiones finales.

Gripper. En este dominio el planificador siempre nos retornó el plan óptimo por lo tanto era imposible obtener un mejor plan, solo podíamos obtener el mismo plan con un mejor tiempo. Pero al ser un problema tan sencillo para resolver el planificador siempre tardó 0.00 segundos en resolver el problema.

En conclusión podemos decir que si bien no hubo una mejora en el rendimiento del planificador. El Split con gamma 0.5 recuperó en todos los casos el plan óptimo y mantuvo el rendimiento en tiempo.

Vale la pena mencionar el comportamiento del robot sobre el dominio spliteado con gamma 0.3. Ya que en el plan despliteado podemos ver que el robot siempre trabaja con un solo brazo, es decir que nunca toma dos pelotas, viaja y las deja a ambas.

Por lo tanto hasta ahora solo podemos concluir que hay algunos splits que funcionan mejor que otros. Pero no hemos podido optimizar el rendimiento del planificador sobre el dominio original.

Cards. El algoritmo de Split no pudo encontrar un split intermedio entre 1 y 7 acciones, esto provocó que comparemos el dominio original contra el atom split. Analizando Gripper vimos que algunos splits funcionan mejor que otros, por lo tanto comparar el dominio original contra el máximo split posible es injusto.

El mayor problema de este dominio se encuentra en el tiempo de búsqueda, mientras que podemos ver que no tiene problemas de instanciación ya que solo genera instancias fáciles.

Por lo tanto podemos concluir que Split con un gran número de acciones, no se comporta muy bien con dominios que tienen problemas en la búsqueda, ya que Split siempre incrementa el número de estados al dividir las acciones.

River Game. Al no ser posible encontrar un Split sobre este dominio es inútil analizarlo. Pero es interesante para mostrar que a pesar de ser un dominio prometedor, ya que tiene una máxima interfaz de cuatro parámetros, no siempre es posible aplicar la técnica de Split.

Checkers. Podemos ver que en este dominio el algoritmo de Spit optimiza el tiempo y mantiene la calidad de los planes encontrados por el planificador. Y además logra que el planificador termine en muchos problemas donde no podía hacerlo sobre el dominio original.

Nuevamente solo un Split encontrado funcionó bien sobre este dominio y fue un Split que no generó una gran cantidad de acciones nuevas, sino solo dos más.

Además este dominio no tiene problemas de búsqueda pero si de instanciación. Por lo tanto podemos concluir que esta clase de dominio es óptima para utilizar el algoritmo de Split.

En general podemos concluir que no existe un gamma único que siempre funcione, sino que hay que buscar cual es el gamma adecuado para cada dominio. Pero tampoco hay garantías de que exista dicho gamma.

En todos los dominios se redujo la máxima interfaz al dividir las acciones lo suficiente. Siempre se logró reducir la cantidad de instancias generadas y por lo tanto también el tiempo de instanciación. Siempre se incrementó la cantidad de estados provocando que el tiempo de búsqueda también se incremente en mayor o menor medida.

Por lo tanto podemos concluir en un primer análisis que la técnica de optimización de dominios que presentamos en esta tesis, funciona muy bien sobre dominios que tienen problemas de instanciación y que no tienen problemas de búsqueda. Estos problemas por lo general son los que tienen acciones con interfaces grandes o un gran número de objetos del mismo tipo. La técnica no funciona bien sobre dominios con problemas en la búsqueda, estos problemas por lo general tienen metas muy distantes al estado inicial, y una gran cantidad de objetos sobre los cuales buscar.

5.2. Sobre la experiencia personal

Personalmente esta tesis me sirvió para aprender a investigar sobre un tema en particular a fondo, estudiar los resultados y poder redactarlo para que otras personas puedan entenderlo.

Creo que hemos obtenido resultados interesantes, principalmente al haber encontrado un dominio donde la técnica funciona muy bien, y haber logrado caracterizar los dominios en los que la técnica es útil.

En conclusión estoy conforme con el trabajo realizado, ya que contribuyó en la investigación que se está llevando a cabo en el grupo de planning del FaMAF. Principalmente el algoritmo de Unsplit, el cual cierra el ciclo necesario para poder realizar tests sobre la técnica de optimización desarrollada. También los nuevos dominios generados pueden ser útiles para el grupo y para una próxima IPC.

En un futuro trabajo sería interesante ampliar los tests realizados incluyendo más dominios con características similares, y verificar si se mantiene el comportamiento observado en este trabajo. También sería importante realizar los tests con otros planificadores que utilicen heurísticas diferentes a la que utiliza FF.

Referencias

1. S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, pp. 3–5, 1995.
2. M. Gelfond and Y. Kahl, *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach.* Cambridge University Press, 2014.
3. M. Sipser, *Introduction to the Theory of Computation.* PWS Publishing, 1997.
4. ICAPS Competitions, *The Seventh International Planning Competition*, 2011.
5. Newell and Simon, "Report on a general problem-solving program," 1959.
6. Fikes and Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," 1971.
7. Penberthy and Weld, "Ucpop: A sound, complete, partial order planner for adl.," 1992.
8. A. Blum and M. Furst, "Graphplan algorithm," 1995.
9. J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.
10. J. Hoffmann. Saarland University - Foundations of Artificial Intelligence Group, Campus E1 1, 66123 Saarbrücken, Germany.
11. M. Helmert, "The fast downward planning system," vol. 26, pp. 191–246, 2006.
12. S. Richter, M. Westphal, and M. Helmert, "Lama 2008 and 2011," 2011.
13. C. Areces, F. Bustos, M. Dominguez, and J. Hoffmann, "Optimizing planning domains by automatic action schema splitting," 2014.
14. D. Mcdermott, M. Ghallab, A. Howe, and et al., "Pddl - the planning domain definition language," 1998.
15. G. van Rossum, B. Warsaw, and N. Coghlan, "Style guide for python code," 2001.