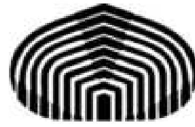

UNIVERSIDAD SIMÓN BOLÍVAR

Ingeniería de Computación

Desarrollo y Evaluación Empírica de Algoritmos de Toma de Decisión Secuencial bajo Incertidumbre

Por

Mariana Bustamante Brito, Simón Enrique Ortiz Branco y Luis Eduardo Ovalle Poveda

Proyecto de Grado

Presentado ante la Ilustre Universidad Simón Bolívar

como Requerimiento Parcial para Optar el Título de

Ingeniero en Computación

Sartenejas, Enero de 2007

UNIVERSIDAD SIMÓN BOLÍVAR

DECANATO DE ESTUDIOS PROFESIONALES

COORDINACIÓN DE INGENIERÍA DE COMPUTACIÓN

ACTA FINAL DEL PROYECTO DE GRADO

DESARROLLO Y EVALUACIÓN EMPÍRICA DE ALGORITMOS DE TOMA DE DECISIÓN SECUENCIAL BAJO INCERTIDUMBRE

Presentado Por:

MARIANA BUSTAMANTE BRITO, SIMÓN ENRIQUE ORTIZ BRANCO
Y LUIS EDUARDO OVALLE POVEDA

Este proyecto de Grado ha sido aprobado por el siguiente jurado examinador:

Prof. Emely Arráiz

Prof. Roger Soler

Prof. Blai Bonet (Tutor Académico)

SARTENEJAS, 29 de enero de 2007

Desarrollo y Evaluación Empírica de Algoritmos de Toma de Decisión Secuencial bajo Incertidumbre

Por

Mariana Bustamante Brito, Simón Enrique Ortiz Branco y Luis Eduardo Ovalle Poveda

RESUMEN

Toma de decisión secuencial en ambientes con incertidumbre es el marco teórico subyacente en diversas áreas de la Inteligencia Artificial tales como, entre otras, planificación de movimiento de robots, planificación y planeación automatizada bajo incertidumbre, modelos de juegos estocásticos, etc. Las tareas de decisión secuencial son entendidas en términos del modelo matemático conocido como Problema de Decisión de Markov (MDP).

El tipo de problemas tratados en Inteligencia Artificial típicamente son descritos utilizando lenguajes de representación en términos de variables proposicionales que combinadas generan miles de millones de estados. En consecuencia, los algoritmos tradicionales para resolver MDPs son generalmente irrealizables para tales problemas excepto para instancias triviales. Sin embargo, recientemente ha habido un número de novedosos algoritmos desarrollados en IA que tratan de explotar características *escondidas* de los MDPs tales como el pequeño número relativo de estados *relevantes*.Algunos de estos algoritmos están basados en búsqueda heurística que explota información sobre el estado inicial dado del sistema y cotas inferiores. Se ha demostrado que estos algoritmos son correctos.

En este trabajo evaluamos varios algoritmos para MDPs basados en búsqueda heurística sobre un amplio conjunto de problemas y los comparamos con los algoritmos clásicos. Para ello implementamos diferentes algoritmos, definimos e implementamos diferentes problemas bajo el marco de los MDPs, y realizamos un gran número de experimentos y analizamos sus resultados. Entre los aportes más relevantes de este trabajo se encuentran: los algoritmos recientes superan a los clásicos cuando la proporción de estados *relevantes* es baja, no hay un dominio de uno de los algoritmos sobre los demás y la función heurística propuesta, ϵ-descendiente, acelera considerablemente la convergencia de LDFS.

Índice general

Índice general III

Índice de Figuras VI

Índice de Algoritmos VII

Glosario de Términos VIII

Capítulo 1. Introducción 1

Capítulo 2. Marco Teórico 4

2.1. Problema de Decisión de Markov 4

2.1.1. Definición 5

2.1.2. Solución 6

2.2. Búsqueda Heurística 7

2.2.1. Heurística Cero 8

2.2.2. Heurística Min-min 8

2.2.3. Heurística Atom-min-backwards 9

2.3. Algoritmos 9

2.3.1. Value Iteration 10

2.3.2. LRTDP 11

2.3.3. ILAO* 14

2.3.4. LDFS 16

2.4. Planificación Probabilística 18

2.4.1. Definición 20

2.4.2. Solución - Un enfoque MDP 21

Capítulo 3. Diseño 22

3.1. Estructura y Funcionamiento - Common 22

3.1.1. Problem 23
3.1.2. Heuristic 23
3.1.3. Hash 24
3.1.4. Hashing 24
3.1.5. Algorithm 24
3.1.6. Utils 24
3.1.7. Ejecución 24
3.2. Estructura y Funcionamiento - mGPT 25
3.2.1. Algoritmos 25
3.2.2. Heurísticas 26
3.2.3. Diseño 27
3.2.4. Ejecución 27
3.3. Limitaciones 28
3.3.1. Common 28
3.3.2. mGPT 30
3.4. Soluciones Propuestas 30

Capítulo 4. Implementación **32**

4.1. Soluciones Implementadas 32
4.1.1. Common 32
4.1.2. mGPT 33
4.2. Dominios implementados 35
4.3. Tron 35
4.3.1. Descripción del problema 35
4.3.2. Descripción de la Solución 35
4.4. MTS (Moving Target Search) 37
4.4.1. Descripción del problema 37
4.4.2. Descripción de la Solución 37
4.5. Spy 38

4.5.1. Descripción del Problema 38
4.5.2. Descripción de la Solución 39
4.5.3. Representación de la Habitación 39

Capítulo 5. Experimentos y Resultados **41**
5.1. Dominios 41
5.2. Heurísticas Utilizadas 41
5.3. Experimentos 41
5.3.1. IPC5 42
5.3.2. Tron 42
5.3.3. MTS 42
5.3.4. Spy 43
5.3.5. Spy Epsilon-descendiente 43
5.4. Análisis de Resultados 43
5.4.1. IPC5 43
5.4.2. Tron 50
5.4.3. MTS 51
5.4.4. Spy 53
5.4.5. Spy Epsilon-descendiente 55

Capítulo 6. Conclusiones y Recomendaciones **57**
6.1. Conclusiones 57
6.2. Direcciones Futuras 58

Bibliografía **60**

Índice de figuras

1. La habitación de donde desea escapar sin ser notado. 1
2. Comparación de algoritmos - heurística 0 - BlocksWorld 43
3. Comparación de algoritmos - heurística min-min-lrtdp - BlocksWorld 44
4. Comparación de heurísticas - BlocksWorld 44
5. Comparación de algoritmos - heurística 0 - Elevators 45
6. Comparación de algoritmos - heurística min-min-lrtdp - Elevators 45
7. Comparación de algoritmos - heurística 0 - Schedule 46
8. Comparación de algoritmos - heurística min-min-lrtdp - Schedule 46
9. Comparación de algoritmos - heurística Backwards - Schedule 47
10. Comparación de algoritmos - heurística 0 - Drive 47
11. Comparación de algoritmos - heurística min-min-lrtdp - Drive 48
12. Comparación de algoritmos - heurística Backwards - Drive 48
13. Comparación de algoritmos - heurística 0 - Zeno 49
14. Comparación de algoritmos - heurística min-min-lrtdp - Zeno 49
15. Comparación de algoritmos - heurística 0 - Tron 50
16. Comparación de algoritmos - heurística min-min-vi - Tron 50
17. Comparación de algoritmos - heurística dist.min - Tron 51
18. Comparación de algoritmos - heurística 0 - MTS 52
19. Comparación de algoritmos - heurística min-min-vi - MTS 52
20. Comparación de algoritmos - heurística cam.min/2 - MTS 53
21. Comparación de algoritmos - heurística 0 - Spy 53
22. Comparación de algoritmos - heurística min-min-vi - Spy 54
23. Comparación de algoritmos - heurística i-Manhattan - Spy 54
24. Comparación de heurísticas - Spy ϵ-descendiente 55
25. Comparación de algoritmos - Spy ϵ-descendiente 56

Índice de Algoritmos

1. Value-Iteration 11
2. Funciones auxiliares de LRTDP 13
3. LRTDP 13
4. ILAO* 15
5. FIND-and-REVISE 16
6. Learning Depth-First Search 18

Glosario de Términos

AO* algoritmos clásico para la solución de problemas modelados como grafos AND/OR.

BNF *Backus-Naur form*

$c(a,s)$ costo de tomar la acción a en el estado s.

$c(s)$ costo de llegar al estado s.

DFS *Depth-First Search*.

ϵ *epsilon*, error entre la esperanza de la solución encontrada y el valor real.

FF *Fast-Forward*

HDP *Heuristic Dynamic Programming*

$h(s)$ valor de la función heurística para el estado s.

ILAO* *Improved Loop AO**

LDFS *Learning in Depth-First Search*

LRTDP *Labeled Real-Time Dynamic Programming*

MDP *Markov Decision Problem*

mGPT *mini General Planning Tool*

PDDL *Planning Domain Description Language*

PPDDL *Probabilistic Planning Domain Description Language*

$P_a(s'|s)$ función de distribución de probabilidades para la trancisión del estado s a s' dada la acción a.

$Q_v(s)$ Qvalue(s).

$Qvalue(s)$ $c(a,s) + \sum_{s' \in S} P_a(s'|s)h(s')$

Capítulo 1

Introducción

Usted es un espía. Se encuentra en una habitación con un guardia que vigila los pasillos por los que usted debe pasar para alcanzar la salida y completar su misión (ver figura 1, Ud. es S, el policía es P, la salida es G, el círculo rojo alrededor de P es el rango de detección del policía). Sobra decir que no quiere que el guardia lo descubra.

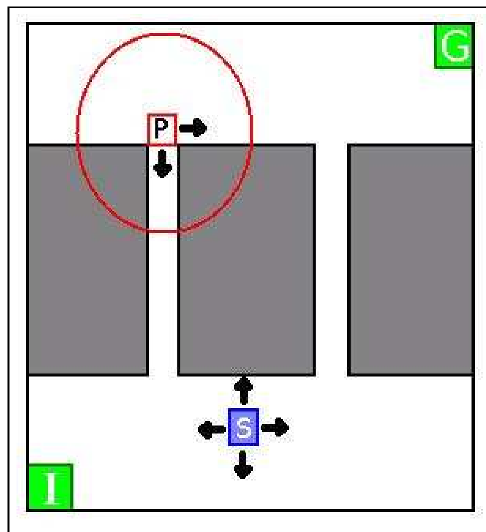

Figura 1: La habitación de donde desea escapar sin ser notado.

Para solucionar exitosamente su problema usted debe construir un plan. Éste debe considerar que el guardia, de vez en cuando, cambia el sentido de su recorrido. Adicionalmente, no cuenta con mucho tiempo para completar su planificación ya que corre el riesgo de ser descubierto.

El plan que usted necesita debe ser de una forma tal que para cada posición suya y del guardia exista una acción óptima a tomar. Esto se conoce como *política*. El problema en el que usted se encuentra inmerso se denomina *Problema de Decisión de Markov* o *MDP* [Rusell y Norvig, 2003].

Sería ideal que usted contara con cierta información acerca de cuáles posiciones de usted y el guardia son más relevantes para ser consideradas en la planificación. A una función que brinde dicha información se le llama *heurística*.

Existe un método para calcular el plan que usted requiere, se conoce como *Value-Iteration* [Rusell y Norvig, 2003]. Está demostrado que siempre funciona, sin embargo, usted se verá obligado a considerar todos y cada uno de las posibles combinaciones de posiciones entre usted y el guardia, ¡terrible! Usted será un muy cansado espía para cuando haya finalizado el proceso.

Existen también otras formas de construir el plan a las cuales llamaremos *algoritmos*. Se sabe que sirven, y se sospecha que son más eficientes que el mencionado *Value-Iteration*, pero no se ha investigado empíricamente su desempeño.

Nosotros, como miembros de su agencia de espionaje, evaluaremos cada una de las opciones disponibles y le haremos saber cuál es la más conveniente en su no tan holgada situación.

Los mencionados algoritmos son *ILAO**, *LRTDP* y *LDFS* [Hansen y Zilberstein, 2001] [Bonet y Geffner, 2003] [Bonet y Geffner, 2006]. A lo largo de este trabajo se realizará por primera vez un estudio comparativo de estos. Para ello esta investigación se estructurará de la siguiente forma: en el capítulo 2, **Marco Teórico**, se hará una introducción a la teoría de *MDPs* y planificación; en el capítulo 3, **Diseño**, se explicará la estructura de nuestra solución en *software*; en el capítulo 4, **Implementación**, se desarrollarán los tópicos concernientes a la implementación de la solución antes expuesta; en el capítulo 5, **Experimentos y Resultados**, encontrará en detalle los experimentos realizados y los resultados obtenidos; finalmente, se cierra con el capítulo 6, **Conclusiones y Recomendaciones**, que consiste de un análisis conclusivo de los datos obtenidos.

Esta investigación no se limitará a estudiar el dominio del problema del espía, se utilizarán dominios representativos utilizados en la evaluación empírica de planificadores participantes de la Competencia Internacional de Planificación *IPC5*.

Entre los aportes más importantes conseguidos a través de esta investigación se encuentran: los algoritmos propuestos se desempeñan mejor que *Value-Iteration* en todos los casos en que la proporción de estados relevantes entre el espacio de búsqueda es pequeña; entre *LRTDP*, *ILAO** y *LDFS* no hay un algoritmo que domine a los demás; la heurística que se propone, ϵ-descendiente, no beneficia a *LRTDP* ni a *ILAO**, pero acelera la convergencia de *LDFS* hasta 10 veces; existen dominios donde *ILAO** escala mejor que los demás algoritmos, en casos iniciales pequeños su desempeño no es sobresaliente, pero a medida que aumenta el espacio de búsqueda supera a sus competidores.

Capítulo 2

Marco Teórico

2.1. Problema de Decisión de Markov

Al problema del cálculo de una política óptima, en un ambiente accesible y estocástico, con base en un modelo de transición conocido se le llama **Problema de Decisión de Markov** [Rusell y Norvig, 2003].

Mejor conocido por sus siglas en inglés **MDPs (Markov Decision Problem)** , deben su nombre a Andrei Markov [1] . De hecho, los problemas de decisión a menudo suelen clasificarse entre aquellos que cumplen la propiedad de Markov y aquellos que no. Dicha propiedad establece que las probabilidades de transición de un determinado estado dependen solamente del estado y no de la historia previa. Esto significa que si un determinado estado de un MDP es conocido en tiempo t, la transición a un nuevo estado en tiempo *t+1* es independiente de todos los estados previos.

De forma más precisa, un MDP modela problemas de toma de decisión secuencial. Estos comprenden la aplicación de acciones que transforman un estado en uno de un conjunto de posibles estados sucesores. Cada uno de estos posibles estados sucesores tiene asociada una determinada probabilidad en la función de transición. Es decir, para un estado s y una acción a, existe una función de transición $P_a(s'|s)$ que determina las probabilidades de transición a los estados sucesores s'[2]. Una vez alcanzado el siguiente estado se computa una recompensa que está asociada al mismo.

[1] Andrei Andreyevich Markov (1856-1922) Matemático ruso conocido por sus trabajos en la teoría de los números y la teoría de probabilidades.

[2] Los números correspondientes a la función $P_a(s'|s)$ son reales y no negativos y su suma sobre los estados $s' \in S$ debe ser 1.

Aunque en general un MDP puede tener infinitos estados y conjuntos de acciones, más adelante nos enfocaremos sólo en resolver problemas para espacios de estados y acciones finitos.

Los problemas de decisión de Markov son una extesión de las cadenas del mismo nombre [3], la diferencia es la adición de acciones (permitiendo elegir) y de recompensas [Meyn y Tweedie, 1993]. De manera que si para cada estado hubiese solamente un acción posible, o esa acción estuviese de alguna manera determinada por la naturaleza del mismo estado, el problema de decisión se reduciría a una cadena de Markov.

A continuación se presenta la definición formal de un MDP, más específicamente de una clase particular de los mismos, correspondiente a los problemas del tipo estocástico del trayecto más corto (*stochastic shortest-path problems*). Este modelo no es más que un MDP con un conjunto de estados terminales. Las acciones en los estados terminales no tienen costo y no producen cambios, es decir, ninguna acción puede causar una transición fuera del mismo estado. De ahora en adelante el término MDP será empleado indistintamente para referirse a esta clase específica del mismo.

2.1.1. Definición

Stochastic shortest-path problem:

M1. un espacio de estados S discreto y finito.

M2. un estado inicial $S_0 \in S$.

M3. un conjunto $G \subseteq S$ de estados terminales.

M4. un conjunto de acciones $A(s) \subseteq A$ aplicables en cada estado $s \in S$.

M5. probabilidades de transición $P_a(s'|s)$ para cada $s \in S$, $a \in A(s)$.

M6. costo de acciones positivos $c(a, s) > 0$, y

M7. estados completamente observables.

[3] Las cadenas de Markov son procesos estocásticos en tiempo discreto que satisfacen la propiedad de Markov.

La solución de un MDP no es una secuencia de acciones, sino una función π que transforma estados s en acciones $a \in A(s)$. Tal función recibe el nombre de *política*. El objetivo es alcanzar un estado terminal incurriendo en el mínimo costo. Una solución óptima para modelos de la forma anteriormente descrita (M1 - M7) siempre existe cuando algún estado terminal es alcanzable desde cada estado $S_0 \in S$ con alguna probabilidad mayor que cero. Sin embargo esta solución óptima puede no ser única en todos los casos.

2.1.2. Solución

Sea V^π la función de evaluación de una determinada política π, se asume que para todo MDP debe existir al menos una política *apropiada* [4] y que, para toda política *no-apropiada* existe al menos un estado con un costo infinito asociado. Es decir, es posible alcanzar un estado terminal partiendo desde cualquier estado y todos los ciclos tienen un costo resultante positivo. Esta asunción corresponde a Bertsekas [Bertsekas, 1995] en el desarrollo de su teoría de los problemas estocásticos del trayecto más corto.

El objetivo es encontrar una política óptima π^* *dominante* [5] sobre cualquier otra política, de manera que su función de evaluación V^* satisface el siguiente sistema de $|S|$ ecuaciones no lineales, siendo S el conjunto de estados finitos correspondientes al MDP:

$$V^*(i) = \begin{cases} 0 \text{ , si } i \text{ es estado terminal,} \\ min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} P_a(j|i) V^*(j) \right] \text{ , si no.} \end{cases}$$

La anterior es denominada *ecuación de optimalidad de Bellman*. Algoritmos de programación dinámica computan la función de evaluación que satisface la ecuación mediante sucesivas estimaciones y mejoras al valor de V^* a través de continuas actualizaciones.

Tradicionalmente los métodos de programación dinámica están pensados para calcular políticas completas. Sin embargo, algoritmos más modernos buscan computar políticas

[4] Se considera que una política es *apropiada* si para todo estado es alcanzable un estado terminal con probabilidad 1.0.

[5] Una política π es dominate sobre una política π' si $V^\pi(i) \leq V^{\pi'}(i)$ para todo estado i.

parciales sobre un conjunto de estados más pequeños que podrián clasificarse como relevantes. Esto se logra a través del uso de las llamadas *funciones heurísticas* que proveen información sobre el costo estimado de alcanzar un estado terminal desde cualquier estado s .

2.2. Búsqueda Heurística

Una *heurística* es una función que se utiliza para aproximar el costo de un estado determinado en un problema de búsqueda.

Se denomina *búsqueda heurística* a la técnica de búsqueda informada que utiliza una función heurística para estimar el costo de llegar a un estado final a partir de un estado cualquiera del problema. De esta forma, es posible generar buenas soluciones que sólo requieran visitar una parte de los posibles estados existentes.

Para que una heurística funcione de forma óptima es necesario que produzca valores cercanos a los reales para cada estado del problema, sin llegar a sobreestimarlos. Si una función heurística cumple con ésta característica se le llama *admisible*. [Rusell y Norvig, 2003]

Todas las heurísticas utilizadas en los experimentos que se describirán son admisibles.

El valor de la función heurística para todos los estados puede ser calculado antes de comenzar a ejecutar el algoritmo de búsqueda, o durante la ejecución del mismo cada vez que sea necesario.

Una de las técnicas más utilizadas para crear funciones heurísticas consiste en disminuir el número de restricciones del problema, con el objetivo de obtener un *problema relajado*, donde el valor de cada estado es más fácil de calcular.

A continuación se describen las heurísticas más importantes utilizadas durante esta investigación.

2.2.1. Heurística Cero

Utilizada únicamente como método de comparación. Consiste en colocar el valor cero como costo estimado para todos los estados del problema.

2.2.2. Heurística Min-min

La heurística *Min-min* [Bonet y Geffner, 2001] se obtiene a partir de relajar el problema para llevarlo desde un problema probabilístico, descrito por la siguiente ecuación de Bellman

$$V^*(s) \stackrel{def}{=} min_{a \in A(s)} \Big[c(s,a) + \sum_{s' \in S} P_a(s'|s) V^*(s') \Big],$$

hasta un problema determinístico de cálculo del camino más corto con una ecuación de Bellman de la forma,

$$V^*_{min}(s) \stackrel{def}{=} min_{a \in A(s)} c(s,a) + min\{V^*_{min}(s') : P_a(s'|s) > 0\}.$$

De esta manera se ha convertido cada operador probabilístico a con n posibles efectos en n operadores determinísticos a_i con un único efecto con probabilidad 1.0 de ocurrir.

Debido a esta conversión se puede elegir el efecto más conveniente del operador de manera que la función heurística se mantenga admisible.

Finalmente, el problema ya relajado puede ser resuelto mediante cualquier algoritmo de búsqueda de caminos. Durante la presente investigación se utilizó el algoritmo *LRTDP* [Bonet y Geffner, 2003] para calcular la heurística; dado que este algoritmo también necesita de una heurística para funcionar correctamente se escogieron las heurísticas *Cero* y *Backwards-1*.

2.2.3. Heurística Atom-min-backwards

La heurística *Atom-min-backwards* [Bonet y Geffner, 2001], es una función heurística computada a partir del problema modelado en STRIPS [Fikes y Nilsson, 1971].

STRIPS es un lenguaje basado en la lógica proposicional formado por un conjunto de proposiciones o *átomos* A, usados para representar los estados del problema de la siguiente forma:

- El estado inicial $S_0 \subseteq A$
- Los estados objetivos $G \subseteq A$
- Operadores $O = (prec, add, del)$ donde $add \subseteq A$, $del \subseteq A$ y $add \cap del = \emptyset$

El resultado de aplicar un operador O sobre un estado S es:

$$Res(S, O) = \begin{cases} \bot \text{ , si } prec \nsubseteq S \\ (S \backslash del) \cup add \text{ , si } prec \subseteq S \end{cases}$$

Búsqueda Inversa

Para calcular el valor de la función heurística para cada estado del problema relajado se utiliza el método de *búsqueda inversa* que comienza a partir del estado objetivo y finaliza en el estado inicial. En la *búsqueda inversa* los estados pueden entenderse como *conjuntos de estados subterminales*, en los cuales la “aplicación” de una acción en un estado terminal produce una situación en la que la ejecución de la acción permite alcanzar este estado objetivo.

2.3. Algoritmos

En esencia el objetivo de los algoritmos es resolver la ecuacion de Bellman para un dominio dado.

Los algoritmos utilizan un esquema itereativo de aproximaciones sucesivas para resolver dicha ecuación. En general, la ecuación no se puede resolver de maneara exacta, por eso introducimos la noción de residual (ϵ, error entre la solución encontrada y el valor real) y lo que se quiere es encontrar una solucion con un residual dado.

En la práctica no hace falta resolver la ecuación para todos los estados. Es suficiente con resolverla para un subconjunto de estados que sea cerrado bajo la política *greedy*[6] y que contenga al estado inicial s_0. Esto da origen a algoritmos como LRTDP, ILAO* y LDFS.

2.3.1. Value Iteration

Value Iteration es el algoritmo clásico utilizado para el cálculo de una política óptima en un problema modelado como MDP [Rusell y Norvig, 2003]. Por tanto se utilizará como base para la evaluación de otros algoritmos. La esencia de *Value Iteration* es resolver iterativamente el conjunto de ecuaciones de Bellman del problema.

Para cada estado en un MDP se tiene asociada una ecuación de Bellman cuya incógnita es la utilidad del estado en cuestión [Rusell y Norvig, 2003]. Este valor depende del valor de las utilidades de los estados sucesores. Para poder hallar el valor de la utilidad de un estado es necesario resolver simultáneamente el sistema de ecuaciones. Dado que la ecuación de Bellman es no lineal (la operación *min* no es lineal) no se puede resolver a través del álgebra lineal. En cambio, se utiliza un enfoque iterativo: inicialmente las utilidades de cada estado poseen valores admisibles, para cada estado se calcula el nuevo valor de su utilidad aplicando la actualización de Bellman:

$$U_{i+1}(s) := c(s) + \min_{a} \sum_{s'} P_a(s'|s) U_i(s')$$

[6] Una función de valores o una heurística h define una política *greedy* π_h de la siguiente manera:

$$\pi_h(s) = \text{argmin}_{a \in A(s)} c(a, s) + \sum_{s' \in S} P_a(s'|s) h(s')$$

donde $U_i(s)$ es la utilidad del estado s en la iteración i con $c(s)$, γ y $P_a(s'|s)$ tal como se definieron anteriormente. Estas iteraciones se efectúan hasta que la diferencia del valor para cada estado entre dos iteraciones sea menor o igual que un valor ϵ dado.

Realizar una actualización para cada estado en el conjunto de estados posibles se conoce como *actualización de programación dinámica*.

El pseudo-código se puede ver en el algoritmo 1.

```
Value-Iteration(S : set of states)
repeat
    U := U' ; δ := 0
    foreach state s in S do
        U'(s) := R(s) + mín_a Σ_s' P_a(s'|s)U(s')
        if |U'(s) − U(s)| > δ then
            δ := |U'(s) − U(s)|
until δ ⩽ ε
```

Algoritmo 1: Value-Iteration

Está demostrado que este algoritmo converge a una solución única y óptima en un tiempo finito dado un ϵ [Rusell y Norvig, 2003].

2.3.2. LRTDP

La esencia de *LRTDP*, o *Labeled Real-time Dynamic Programming*, consiste en simular corridas (que llamaremos *trials*) a partir del estado inicial s_0 hasta llegar a un estado etiquetado como *solved* realizando *actualizaciones de programación dinámica* en cada estado visitado [Bonet y Geffner, 2003]. Inicialmente sólo los estados terminales están etiquetados como *solved*. Las acciones que se escogen son *greedy* dada la política actual. El estado sucesor s' consecuencia de haber aplicado la acción a en el estado s se escoge según la probabilidad que tenga de aparecer dada la función de transición $P_a(s'|s)$.

La idea detrás del modelo de etiquetamiento de LRTDP es saber cuáles estados han convergido a su valor real para evitar visitarlos de nuevo. Esto permite explorar con

más frecuencia caminos menos probables que son necesarios para la convergencia de la solución [Bonet y Geffner, 2003].

El método de etiquetamiento se conoce como *CheckSolved*. Básicamente, se etiqueta un estado s como resuelto si sólo si s y todos sus sucesores han convergido. *CheckSolved* busca en el grafo *greedy*[7] con raíz s un estado cuyo residual sea mayor que ϵ. Si y sólo si no se consigue dicho estado se etiqueta s y todos los estados en su grafo *greedy* como resueltos. Etiquetar un estado como resuelto significa que ha convergido, y por definición, cuando un estado s converge también convergen los estados en su grafo *greedy*. En el caso en que dicho estado haya aparecido, el algoritmo realiza actualizaciones a éste y todos los demás estados no convergentes que se hayan encontrado. La exploración no sigue por debajo de estados como éstos.

LRTDP al terminar el *trial* invoca a *checkSolved* en orden inverso a como se visitaron los estados no *solved* mientras no se encuentren estados que no hayan convergido. El algoritmo termina cuando el estado raíz s_0 se etiqueta como *solved*.

El pseudo-código de LRTDP se encuentra en el algoritmo 3.

[7] grafo que contiene los estados alcanzables desde el estado s con la política *greedy*.

```
state :: pickNextState(a : action)
begin
  | escoger s' con probabilidad P_a(s'|s)
  | return s'
end

state :: greedyAction()
begin
  | return argmin_{a∈A(s)} s.Qvalue(a)
end

state :: update()
begin
  | a = s.greedyAction()
  | s.value = s.Qvalue(a)
end
```

Algoritmo 2: Funciones auxiliares de LRTDP

```
LRTDP(s : state, ε : float)
begin
  | while ¬s.solved do
  |   └ LRTDPTrial(s, ε)
end

LRTDPTrial(s : state, ε : float)
begin
  | visited := emptyStack
  |
  | while ¬s.solved do
  |   | visited.push(s)
  |   |
  |   | if s.goal() then
  |   |   └ break
  |   | a = s.greedyAction()
  |   |
  |   | s.update()
  |   |
  |   └ s.pickNextState(a)
  |
  | while visited ≠ emptyStack do
  |   | s = visited.pop()
  |   |
  |   | if ¬checkSolved(s, ε) then
  |   └   └ break
end
```

Algoritmo 3: LRTDP

2.3.3. ILAO*

*ILAO**, o *Improved Loop AO**, es una versión mejorada el algoritmo LAO, que a su vez es una generalización del algoritmo AO* para MDPs [Hansen y Zilberstein, 2001].

AO* es un algoritmo de búsqueda heurística bien conocido que permite hallar soluciones en forma de grafo acíclico a problemas modelados como grafos AND/OR. El algoritmo se puede dividir en tres partes: *expansión hacia adelante*, en donde se utiliza una heurística para obtener una búsqueda enfocada al escoger un estado más prometedor para la expansión; *revisión de costos*, donde se utiliza programación dinámica para propagar los costos de los nuevos estados hacia la raíz del grafo solución; *etiquetamiento*, donde se etiqueta como *solved* los estados *goal* y aquellos donde todos sus sucesores están etiquetados como *solved*, evitándose así la exploración debajo de estos estados.

Se puede representar un MDP como un grafo AND/OR donde los nodos OR representan la escogencia de la acción a ejecutar y los nodos AND el resultado estocástico consecuencia de la acción. Sin embargo, AO* no se puede utilizar para solucionar MDPs debido a que la parte de *revisión de costos* asume que la solución es un grafo acíclico. Hansen y Zilberstein [Hansen y Zilberstein, 2001] crearon una generalización de AO* que permite hallar soluciones en forma de grafo cíclico conocido como *Loop AO**.

La clave en la generalización de AO* a LAO* consiste en reconocer que el paso de *revisión de costos* en AO* es programación dinámica. Sólo se necesita adaptar este paso para poder generar una solución cíclica. La adaptación consiste en utilizar un algoritmo de programación dinámica para MDPs, como *Value-Iteration*, que tenga el mismo efecto de actualizar los costos de los estados en AO*.

ILAO* surge de la pobre eficiencia de LAO*. Expandir sólo un estado en la frontera del mejor grafo solución por vez, y aplicar varias iteraciones de *Value-Iteration* entre cada

expansión de estados resulta en muchos estados siendo actualizados muchas veces antes de que converjan [Hansen y Zilberstein, 2001].

Una de la soluciones para aumentar la eficiencia consiste en expandir todos los estados en la frontera del mejor grafo solución antes de aplicar el paso de *revisión de costos* que es mucho más costoso a nivel de cómputo [Hansen y Zilberstein, 2001]. La propuesta de Hansen y Zilberstein consiste en explorar el grafo con DFS antes de aplicar el paso de *revisión de costos*.

El esquema del algoritmo se encuentra en el algoritmo 4.

1. El grafo explícito G' consiste sólo del estado inicial s_0
2. *Expandir la mejor solución parcial, actualizar costos y marcar las mejores acciones*

   Mientras el mejor grafo solución contenga un estado *tip* no terminal aplicar DFS en el mejor grafo solución. Para cada estado visitado s aplicar en post-orden:

   I. Si el estado s no está expandido, expandirlo

   II. Asignar $V(s) := min_{a \in A(s)}[c(a,s) + \sum_{s'} P_a(s'|s)V(s')]$ y marcar la mejor acción para s. Resolver empates arbitrariamente.
3. *Test de convergencia*

   Aplicar *Value-Iteration* en los estados en el mejor grafo solución hasta que una de las siguientes condiciones ocurra:

   (i) si el error baja de ϵ, ir al paso 4.

   (ii) si el mejor grafo solución tiene un nuevo estado *tip* no expandido, ir al paso 2.
4. Devolver un grafo solución ϵ-óptimo.

Algoritmo 4: ILAO*

2.3.4. LDFS

LDFS, o *Learning in Depth-First Search*, es un algoritmo de búsqueda informada que aprovecha los beneficios de la programación dinámica y el poder de las técnicas de búsqueda heurística. Consiste en combinar búsquedas en profundidad iterativas con aprendizaje, puede reducirse a IDA* con tablas de transposición en modelos determinísticos, y es también capaz de resolver, con pequeñas variaciones, modelos no-determinísticos, probabilísticos y de árbol de juego. [Bonet y Geffner, 2006] [Bonet y Geffner, 2005a]

El objetivo del algoritmo es computar una política π que minimice el valor del costo $V^*(s_0)$, que resuelve la ecuación de Bellman.

Una función V es una solución a la ecuación de Bellman, y por lo tanto es igual a V^* si el residual es eliminado en todos los estados. Sin embargo, dado un estado inicial s_0 *fijo*, no es necesario eliminar todos los residuales para asegurar la optimalidad.

Se asume un orden en las acciones y se tomará π_V como la política *greedy* obtenida al seleccionar en cada estado la acción *greedy* mínima con respecto a este orden. Si $\epsilon = 0$ y la función V inicial es *admisible*, entonces la política *greedy* π_V resultante es óptima.

LDFS se basa en el algoritmo 5, *Find-and-Revise*.

starting with an admissible V
repeat
  FIND s reachable from s_0 and π_V with $Residual_V(s) > \epsilon$
  Update $V(s)$ to $min_{a \in A(s)} Q_V(a, s)$
until *no such state is found*
return V

Algoritmo 5: FIND-and-REVISE

LDFS implementa la operación *FIND* como una búsqueda en profundidad que considera todas las acciones *greedy* en un estado, utiliza *backtracking* en estados inconsistentes, y actualiza no sólo éstos estados, sino también sus ancestros.

La búsqueda en profundidad es lograda mediante dos ciclos: el primero sobre las acciones *greedy* $a \in A(s)$ en s, y el segundo, anidado, sobre los posibles sucesores $s' \in F(a, s)$, donde F es una función que mapea estados no terminales s y acciones a en *conjuntos* de estados $F(a, s) \subseteq S$. Los nodos finales en esta búsqueda son los estados s inconsistentes, donde se cumple $Q_V(a, s) > V(s)$ para todas las acciones, los estados terminales y los estados marcados como *solved*.

Un estado s es marcado *solved* cuando la búsqueda en sus ancestros no encontró ningún estado inconsistente, esto se maneja mediante el booleano *flag*. Si s es consistente y *flag* se cumple después de realizar una búsqueda en los sucesores $s' \in F(a, s)$ de una acción *greedy* a, entonces s es marcado como *solved*, $\pi(s)$ ahora es a y no se intentarán más acciones sobre s. Si *flag* no se cumple se intenta con la próxima acción *greedy*; si no resta ninguna, s se actualiza.

El pseudo-código de *LDFS* se encuentra en el algoritmo 6.

Optimalidad

Se sabe que una solución óptima es aquella que minimiza $V_\pi(s_0)$. Si π también minimiza $V_\pi(s)$ para todos los estados s alcanzables desde s_0, se dice que π es **globalmente óptima**. *LDFS* y *FIND-and-REVISE* sólo computan políticas globalmente óptimas, ya que continúan actualizando V hasta eliminar *todas* las inconsistencias sobre los estados alcanzables desde s_0 utilizando la política *greedy* π_V.

MDPs

Un problema modelado como MDP puede presentar ciclos, en estos casos ya no es correcto marcar un estado como *solved* cuando la variable *flag* indica que todos los descendientes de s son consistentes (es decir, sus residuales son no mayores que ϵ); ya que pueden existir muchos ancestros alcanzables desde s a través de un ciclo con descendientes aún no explorados.

```
LDFS-DRIVER(s_0)
begin
    repeat solved := LDFS(s_0) until solved
    return (V, π)
end

LDFS(s)
begin
    if s is SOLVED or terminal then
        if s is terminal then V(s) := c_T(s)
        Mark s as SOLVED
        return true

    flag := false
    foreach a ∈ A(s) do
        if Q_V(a, s) > V(s) then continue
        flag := true
        foreach s' ∈ F(s, a) do
            flag := LDFS(s')&[Q_V(a, s) ≤ V(s)]
            if ¬flag then break
        if flag then break

    if flag then
        π(s) := a
        Mark s as SOLVED
    else
        V(s) := min_{a∈A(s)} Q_V(a, s)
    return flag
end
```

Algoritmo 6: Learning Depth-First Search

Para que *LDFS* sea capaz de manejar MDPs se deben realizar dos consideraciones: El valor del residual permitido ϵ deberá ser mayor que *cero* para evitar convergencia asintótica, y debe mantenerse un mecanismo recordatorio para evitar ciclos y reconocer estados *solved*.

2.4. Planificación Probabilística

Los problemas más simples de planificación están dirigidos a encontrar una secuencia de acciones, de forma tal, que para algún estado inicial dado sea posible alcanzar un

estado terminal. Sólo acciones de tipo determinístico son consideradas para tal efecto. Esto implica que partiendo de un determinado estado inicial, y al aplicarse una secuencia de acciones cualquiera, el resultado o estado de transición final resultante es completamente predecible.

La planificación probabilística es una extensión del modelo **no-determinístico** de planificación, donde, aunque no se sabe con certeza el efecto de aplicar una determinada acción a un estado, sí se posee cierto grado de información. Ésta se encuentra referida a la probabilidad con que ocurrirán una diversa serie de eventos.

Dicha información resulta de suma importancia en la formulación de un **plan** solución. Las probabilidades asociadas a la funciones de transición a nuevos estados son importantes al momento de cuantificar los costos y posibilidades de éxito del plan cuando se trata de un dominio no-determinístico. En este caso, por la naturaleza del problema, en muchas ocasiones no basta sólamente con la formulación del plan. Es importante también que sea eficiente en términos del costo de las acciones, y que a su vez tenga la propiedad de máximizar la probabilidad con que los estados terminales o meta serán alcanzados.

Un dominio correspondiente a planificación probabilística está especificado básicamente por un conjunto de estados, un conjunto de acciones, un estado inicial, y un conjunto de estados terminales. También, por supuesto, una distribución de probabilidades. En general, la salida de un algoritmo de planificación es un controlador para el dominio en cuestión, cuyo objetivo es alcanzar la meta con una alta probabilidad. Un plan es un *programa* que toma información de los subsecuentes estados y produce acciones como salida. Los planes se clasifican por su tamaño y su horizonte, y estos son medidores también de la eficiencia del mismo. El tamaño está referido al número de estados considerados en el plan, y el horizonte se refiere al número de acciones necesarias para llegar a un estado terminal.

2.4.1. Definición

1. un conjunto finito de estados S.

2. un estado inicial.

3. un conjunto G de estados terminales, $G \subseteq S$.

4. un conjunto O finito de acciones o que son funciones parciales que transforman cada estado en una distribución de probabilidades sobre S.

5. una distribución I de probabilidades sobre S.

Una acción o es aplicable en aquellos estados para los cuales $o(s)$ está definida. Dichos estados se denotan de la siguiente forma : $prec(o) = \{s \in S|o(s)$ está definida $\}$. $O(s)$ no es vacío para todo $s \in S$.

$o \in O$ son funciones parciales, esto significa que una acción no asocia necesariamente cada estado con una distribución de probabilidad. El motivo es que una acción no es necesariamente aplicable para todo estado.

Los estados están expresados como un conjunto de literales ciertos o falsos en el contexto del cálculo proposicional. El estado inicial también se representa como un conjunto de literales. Las acciones no solo conllevan una serie de precondiciones $prec(o)$ que deben cumplirse antes de poder ser ejecutadas, sino también un conjunto de efectos que éstas tendrán en el estado y que denotaremos como $Del(o) + Add(o)$. Tanto las precondiciones como los efectos son también un conjunto de literales.

De forma más precisa podemos redefinir el problema de planificación probabilístico como una tupla $P = \{A, O, I, G\}$ donde A es un conjunto de literales, O es un conjunto de operadores, y $I \subseteq A$, $G \subseteq A$ son los estados iniciales y terminales. El espacio de estados está determinado por P y es una tupla $S = \{S, S_o, S_G, A, f, c\}$ donde:

1. los estados $s \in S$ son colecciones de literales de A.

2. el estado inicial S_o es I.

3. los estados terminales $s \in S_G$ son tales que $G \subseteq s$.

4. las acciones $a \in A(s)$ son operadores $op \in O$ tales que $prec(op) \subseteq s$.

5. una función de probabilidades de transición f que transforma estados s en estados $s' = s - Del(a) + Add(a)$.

6. los costos de las acciones $c(a)$ son todos iguales a 1.

2.4.2. Solución - Un enfoque MDP

Los problemas de planificación del tipo probabilístico son a menudo modelados a través del *problema de decisión de Markov*. Esto debido al interés de la comunidad del área de inteligencia artificial de usar la tecnología ya existente para resolver MDPs y aplicarla para resolver los problemas de planificación que envuelven cierta incertidumbre.

Básicamente, un MDP es un espacio de estados donde las transiciones entre los mismos son de naturaleza estocástica. De manera abstracta, y en forma simple, éste es el mismo modelo que soporta la planificación probabilística. La mayor diferencia radica en la manera en que los estados y las acciones son representados. Mientras un MDP hace uso de una representación más explícita de estados y acciones, son propias del problema de planificación representaciones más complejas, donde los estados son conjuntos de átomos y las acciones son operadores con precondiciones y efectos.

En general, un problema de planificación con transiciones basadas en sistemas de probabilidades puede ser llevado a un *problema de decisión de Markov*, permitiendo explotar el uso y los beneficios de una serie de algoritmos que han probado ser útiles al momento de hallar soluciones óptimas.

Capítulo 3

Diseño

La implementación inicial de las clases y algoritmos descritos en esta sección fue realizada por los profesores Blai Bonet y Héctor Geffner, y se modificó durante este trabajo para alcanzar los objetivos de la investigación.

El código con el cual se trabajó está dividido en dos partes: **Common** y **mGPT**, ambas se encuentran descritas a continuación.

3.1. Estructura y Funcionamiento - Common

Common es una librería que contiene un conjunto de utilidades que permiten resolver problemas del tipo MDP. La misma está diseñada de forma tal que una vez modelado el problema, éste pueda ser resuelto bajo distintas combinaciónes de heurísticas y algoritmos.

En *Common* se definen diversas secciones, cada una dispuesta con el fin de modelar los diferentes componentes que conlleva el proceso solución. Partiendo desde el mismo modelado y codificación del problema, el conjunto de algoritmos que podrán ser aplicados al mismo, las heurísticas existentes, y mecanismos para especificar nuevas según sea el caso, llegando hasta las estructuras que permitirán el almacenamiento, mesura y acceso a los datos involucrados en el proceso. Este diseño modular hace de *Common* una librería flexible con facilidades para ser modificada, extendida o sencillamente utilizada. En este sentido, para disponer de las funcionalidades de la librería y aplicarlas a un determinado problema, basta con codificar el mismo siguiendo ciertas convenciones dadas por la propia librería. De esta manera, es innecesario realizar cualquier modificación a los algoritmos, heurísticas, etc., que puedan ser causadas por la naturaleza del problema. Esto dado que la codificación del mismo está diseñada para ser independiente de los otros componentes de la librería.

El diseño modular de *Common* está sustentado en las siguientes clases: *Problem*, *Heuristic*, *Hash*, *Hashing*, *Algorithm*, y *Utils.* Cada una de ellas con una funcionalidad perfectamente demarcada descrita a continuación:

3.1.1. Problem

A fin de resolver un MDP a través de las funcionalidades de esta librería es necesario codificar el problema siguiendo los parámetros del tipo *problem*. Es realmente la única etapa del proceso que requiere un entendemiento del funcionamiento del código, superada la misma los demás aspectos son transparentes. En esencia todo *problem* debe tener definido el número de acciones posibles, costo y recompensas asociadas a las mismas. Se requiere además específicar una función de transición y otra de terminación, es decir, que permita saber cuando un estado es o no terminal.

Básicamente, luego de definir estos aspectos del problema en la manera establecida por la librería, no hace falta mucho más para la ejecución del programa y la obtención de resultados.

3.1.2. Heuristic

Envuelve los procedimientos a través de los cuales es computado el valor de la heurística para un determinado estado. Cuenta con tres funciones heurísticas preestablecidas y aplicables a todo problema en general. Estas son: *zero*, *minmin* y *hdp(0)*; más adelante veremos que *hdp(0)* no es de mayor interés en este estudio por lo que no se hará mayor referencia a las misma.

El tipo *heuristic* permite además definir heurísticas específicas para un problema dada la naturaleza del mismo. De esta forma si queremos resolver el popular problema de *n-puzzle* es posible incluir en nuestra codificación del mismo la heurística *manhattan* y hacer uso de la misma.

3.1.3. Hash

Aquí se encuentra definido el tipo de dato básico que reune toda la información asociada a la búsqueda correspondiente a un estado. Etiquetamientos, conteos, incrementos, actualizaciones, etc., de valores son almacenados en este tipo.

3.1.4. Hashing

Modela toda la estructura correspondiente a las tablas de hash. Abarca los procedimientos de búsqueda, almacenamiento y obtención de valores.

3.1.5. Algorithm

Dentro de esta clase están especificados todos los algoritmos que brinda la librería. Entre ellos encontramos *Value-Iterationi, LRTDP, uLRTDP, bLRTDP, eLRTDP, ILAO*, check-solved, hdp-i, hdp, LDFS, LDFS+*. De los cuales destacan *Value-Iteration, LRTDP, ILAO** y *LDFS*. El resto son variaciones de los anteriores o escapan al alcance de esta investigación.

3.1.6. Utils

Comprende los procedimientos que, aunque no forman parte esencial del proceso solución, aportan información de interés al mismo. Aquí se puede establecer, por ejemplo, el nivel de verbosidad de una determinada ejecución, así como también medir el tiempo de corrida empleado por la misma, tanto a nivel de cálculo heurístico como el tiempo consumido por la ejecución del algoritmo en sí.

3.1.7. Ejecución

Dado que *Common* es una librería y no un programa con un componente ejecutable, no existe una forma predefinida de uso y ésta estará sujeta al componente invocador. De cualquier forma, *Common* permite variar los siguientes parámetros: algoritmo, heurística, valor ϵ que servirá para calcular una función de valores ϵ-consistente, verbosidad, etc.

3.2. Estructura y Funcionamiento - mGPT

mGPT es un *planner* diseñado para resolver MDPs especificados en el lenguaje PPDDL. Computa heurísticas basadas en relajaciones del problema a resolver y utiliza diversos algoritmos de búsqueda heurística para determinar una función de valores V ϵ-consistente para los estados alcanzables desde el estado inicial con la política *greedy* basada en V.

Planning Domain Description Language, o PDDL, es un lenguaje BNF extendido introducido en 1998 en la primera Competencia Internacional de Planificación con el propósito de estandarizar el modelaje de los dominios de los problemas de planificación. Este lenguaje se ha convertido en el estandar para intercambio de dominios de planificación, facilitando la evaluación empírica de planificadores. PPDDL, o *Probabilistic PDDL*, es la extensión de PDDL que incluye efectos probabilísticos y recompensas en los estados terminales. Fue introducido para la cuarta Competencia de Planificación en 2004, permite modelar MPDs. La exposición de la sintaxis y semántica de PPDDL y PDDL escapan del alcance de este trabajo. Ver [Younes y Littman, 2004] y [Fox y Long, 2003] respectivamente.

3.2.1. Algoritmos

mGPT posee dos grupos de algoritmos: aquellos que permiten escoger una acción en tiempo real, y aquellos que determinan una función de valores V ϵ-consistente para los estados alcanzables desde el estado inicial s_0 con la política *greedy* basada en V.

El primer grupo está compuesto por el algoritmo *Action Selection for Planning* o ASP [Bonet et al., 1997]. Este algoritmo es básicamente RTDP con *look-ahead*. ASP realiza actualizaciones sobre el valor de los estados de modo que no queda atrapado en ciclos. La política que calcula no es óptima pero sí propia. Este algoritmo no trata de resolver el MDP como tal, sólo trata de escoger una acción en tiempo real luego de poco procesamiento.

En el segundo grupo se encuentran los algoritmos *Value-Iteration*, *LRTDP* y *Heuristic Dynamic Programming* o *HDP*. Ya se han expuesto *VI* y *LRTDP*. *HDP* realiza búsquedas de

estados ϵ-inconsistentes actualizando sus valores; la búsqueda se realiza en DFS sobre el conjunto de estados alcanzables desde s_0 con la política *greedy*. Luego se aplica un esquema de etiquetamiento basado en el *procedimiento de componentes fuertemente conexos de Tarjan* [Bonet y Geffner, 2005b].

Cabe destacar que ASP y HDP no son de interés para este estudio, es por ello que no se han expuesto en profundidad.

3.2.2. Heurísticas

mGPT posee dos grupos de heurísticas basadas en dos formas de relajación: la relajación min-min y la relajación STRIPS. La relajación min-min define un problema de tipo *Shortest Path Problem* determinístico en el espacio de búsqueda original. La relajación STRIPS define un problema de tipo *Shortest Path Problem* determinístico en el espacio de los átomos [1].

Ya se ha expuesto a nivel teórico la relajación min-min. El problema determinístico resultante se resuelve a través de IDA* o a través de LRTDP. La heurística se resuelve en tiempo polinomial en el número de estados.

La relajación STRIPS consiste en transformar el problema determinístico de la relajación min-min en un problema STRIPS, y luego computar la heurística utilizando métodos desarrollados en planificación clásica. Estos métodos se ejecutan en tiempo polinomial en el número de átomos, sin embargo, la transformación a STRIPS requiere tiempo y espacio exponencial [Bonet y Geffner, 2005b].

En el grupo de heurísticas STRIPS se encuentran *Atom-min Backwards*, *Atom-min Forward* y *FF*. Ya se expuso *Atom-min Backwards* anteriormente. *Atom-min Forward* calcula el costo de llegar a un conjunto de átomos de cardinalidad fija desde un estado dado. A diferencia de *Atom-min Backwards* los costos se calculan en un procedimiento *hacia adelante*

[1] símbolos proposicionales utilizados en el lenguaje de representación

partiendo del estado dado y termina en un estado terminal [Bonet y Geffner, 2005b]. En *mGPT* se puede asignar la cardinalidad de los conjuntos de *Atom-min Forward* y *Atom-min Backward*. La heurística *FF* es informativa pero no admisible, por ello no es de interés para este estudio. Es una implementación de la heurística utilizada en el planificador *FF* [Bonet y Geffner, 2005b].

3.2.3. Diseño

mGPT está diseñado para participar en la IPC5 (*International Planning Competition 5*). La participación en la competencia está basada en el modelo cliente-servidor: los participantes se conectan como clientes a un servidor de la IPC5, el cliente envía la acción a tomar en el estado inicial del problema en cuestión, el servidor decide cual es el estado sucesor en base a las probabilidades del problema y le indica dicha transición, el cliente decide la siguiente acción, y así sucesivamente hasta alcanzar un estado terminal.

mGPT posee un sistema de apilamiento de heurísticas. Existen heurísticas que utilizan algoritmos de búsqueda heurística para calcular el valor de un estado, por ejemplo `min-min-lrtdp` y `min-min-ida*`. Estas heurísticas pueden mejorar su aproximación si se les provee de una heurística inicial. Por ejemplo, se puede utilizar la heurística `atom-min-backward` como herurística de `min-min-lrtdp`.

Una vez que *mGPT* ha recibido toda la información necesaria, como el problema a resolver, el algoritmo y la heurística a utilizar, *mGPT* intenta conectarse con un servidor de la IPC5, empieza a resolver el MDP con el algoritmo indicado y realiza el intercambio estado por acción con el servidor.

3.2.4. Ejecución

Una ejecución de *mGPT* comienza por la invocación del mismo:

```
planner <option>* <host>:<port> <problem-file> <problem-name>
```

mGPT cuenta con varias opciones, aquellas que nos interesan son asignar el valor de ϵ, definir el algoritmo a utilizar y definir la pila de heurísticas. `-e <epsilon>` permite asignar un valor ϵ que servirá para calcular una función de valores ϵ-consistente. El algoritmo a utilizar se introduce con la opción `-a` seguido del nombre del algoritmo. La pila de heurísticas se introduce con la opción `-h`, se indica la heurística a utilizar y, si se desea, se concatena con la siguiente heurística con el caracter "`|`"; por ejemplo, `-h "atom-min-1-backwards|min-min-lrtdp"`.

Los argumentos `<host>:<port>` indican el servidor y el puerto en el cual está corriendo el servidor de la IPC5 al cual se conectará *mGPT*. `<problem-file> <problem-name>` indican el archivo PPDDL en el cual se encuentra la descripción del problema y cuál instancia del mismo se ha de resolver.

Una invocación típica de *mGPT* puede tener la siguiente forma:

```
planner -e .01 -a vi -h "zero|min-min-lrtdp" serv.exmpl:8000 p1.pddl bw_5
```

3.3. Limitaciones

Los programas en su estado original presentan ciertas limitaciones en su diseño e implementación que se presentan a continuación.

3.3.1. Common

Stochastic Shortest-Path Problem

Common está diseñado para resolver MDPs del tipo *Stochastic Shortest-Path Problem*. Esto implica que los costos de toda acción en cualquier estado es 1 y que el costo de llegar a un estado terminal es 0.

Esto limita los problemas que se pueden resolver con *Common*. En particular, no se puede utilizar para resolver problemas donde se tengan diferentes tipo de estados terminales, unos preferibles sobre otros. Típicamente, un problema de este estilo se modelaría

asignando costos muy pequeños a los estados terminales deseables y costos muy altos para los estados terminales no deseables. Similarmente, los problemas donde diferentes acciones tienen diferentes costos no son compatibles con *Common*.

Acciones Aplicables

La implementación de *Common* asume que, para cualquier problema, en cualquier estado, todas las acciones posibles son aplicables. Existen dominios donde esto no se cumple.

Es posible rodear esta limitación: al escoger una acción a_i no aplicable en un estado s_i hacer que el estado sucesor de aplicar a_i en s_i sea s_i. Sin embargo, esto complica la implementación del dominio.

Heurísticas Pobres

Common sólo incluye las heurísticas *min min*, *hdp* y la posibilidad de que el usuario cree una propia. En comparación con el sistema de pila de heurísticas se evidencia su menor poder. Esto implica una generación de heurísticas menos informativas y de menor calidad.

Implementar Problemas

Como se explicó anteriormente, *Common* es una librería; como tal su uso implica la codificación de los dominios a resolver en C o C++. Dependiendo de la complejidad del dominio esto puede ser engorroso. Dado que PPDDL es un estándar de intercambio de dominios de planificación sería deseable que *Common* permitiese la solución de los dominios en esta codificación de manera directa.

Obtención de Política

Common permite el cálculo de la función de valores V^* para un dominio. Sin embargo, no saca ningún provecho de dicha función. En particular no se calcula la política óptima en base a V^*, esto impide la utilización de *Common* directamente para la solución de

problemas.

3.3.2. mGPT

Conexión a servidor

mGPT está diseñado para interactuar con los servidores de la competencia IPC5. Como tal, siempre espera recibir la dirección de un servidor de la competencia y el puerto al cual conectarse. Además, está diseñado para empezar el intercambio acción-estado con el servidor una vez que ha resuelto el problema con los algoritmos y las heurísticas dadas.

Dada la robustez de *mGPT* es deseable que se pueda utilizar *offline* y que las soluciones sean reportadas al usuario en vez de participar en el intercambio con el servidor IPC5.

Tiempo de algoritmos y heurísticas

En el diseño actual de *mGPT* las métricas de eficiencia están dadas por el hecho de haber resuelto o no el problema y por comparación con otros participantes del concurso.

El uso *mGPT offline* implica que no se poseen métricas de eficiencia adecuadas. En particular, no se sabe el tiempo de solución del problema y el tiempo invertido en el cómputo de la heurística.

Pocos algoritmos

La cantidad de algoritmos disponibles en *mGPT* no es muy variada. Cuenta con un algoritmo de selección de acción aleatoria (sólo con propósitos de evaluación), *Value-Iteration*, *LRTDP*, *ASP* y *HDP*.

3.4. Soluciones Propuestas

Es evidente que tanto *Common* como *mGPT* presentan carencias importantes que limitan el rango de experimentación, así como el aporte de los datos obtenidos a través de la misma. Algunas de estas limitaciones es posible eliminarlas haciendo modificaciones propias del diseño e implementación del programa que las presenta, ya sea *Common* o

mGPT. Sin embargo, existen otras tantas que escapan al alcance del programa en sí, donde el enfoque apropiado para solucionarlas debe estar orientado a aprovechar las bondades que ambas librerias presentan y ver la manera en que puedan ser combinadas para generar una herramienta sin las limitaciones anteriores. Esta solución evita una gran inversion en tiempo de desarrollo al reusarse los componentes ya existentes.

De esta manera, al acoplarse *Common* y *mGPT* quedarían resueltas las limitaciones de pocos algoritmos y heurísticas pobres. Así mismo, sería posible codificar problemas en PPDDL haciendo su implementación mucho más sencilla. Específicamente para *Common* y el problema de las acciones aplicables agregar una función que verifique esta restricción previo a la expansión de un estado. Se incluirá también la funcionalidad de la obtención de la política óptima permitiendo obtener el plan solución.

En lo referente a *mGPT* añadir la opción de un funcionamiento *offline* eliminando la necesidad de interactuar con un servidor que aporte información sobre el estado sucesor, de esta manera se calcularían políticas completas y no parciales, al tomarse en cuenta todos los estados sucesores relevantes. Se incorporarán también métricas para cuantificar el tiempo utilizado para hallar la solución en general; así como el tiempo de algoritmo y heurística en forma independiente.

Capítulo 4

Implementación

4.1. Soluciones Implementadas

4.1.1. Common

Stochastic Shortest-Path Problem

Se realizaron modificaciones en *Common* que permiten a los algoritmos funcionar apropiadamente con problemas que no necesariamente cumplen todas las condiciones de los problemas de tipo *Stochastic Shortest-Path*. Se sabe que la principal diferencia en los problemas viene dada por el hecho de que contienen diferentes tipos de estados terminales, que aportan un costo determinado a la solución final dependiendo de su conveniencia, la solución óptima debe finalizar con el estado terminal de menor costo.

Inicialmente se incluyó en la clase *Problem* una nueva función *reward* que debe ser establecida para cada problema, en ella se debe especificar el costo de cada uno de los tipos de estados terminales. Se realizaron también modificaciones en la función de cálculo de valores de los estados y en cada uno de los algoritmos para incluir la ejecución de esta función cada vez que el estado explorado sea terminal, de esta manera se obtiene el valor correspondiente y se suma al costo total calculado.

Acciones Aplicables

De la misma forma que en el problema anterior, se incluyó otra función a la clase *Problem*, llamada *applicable* que indica si una acción puede aplicarse en un estado determinado del problema. También se modificaron los algoritmos para que nunca se ejecute una acción sin verificar primero que ésta es aplicable de acuerdo con el estado actual.

Heurísticas Pobres

Como se explicó anteriormente, *Common* sólo contiene las heurísticas *zero, minmin* y *hdp(0)*, mientras que *mGPT* contiene una mayor variedad que incluye: *Atom-min Backwards, Atom-min Forward* y *FF*.

Dado que *mGPT* contiene varias heurísticas completas e interesantes para la investigación, se unificó a la librería *Common* con *mGPT* para aprovechar las heurísticas en todos los problemas seleccionados.

Implementar Problemas

Se sabe que el planificador *mGPT* trabaja sobre problemas implementados en el lenguaje *Planning Domain Description Language* o PDDL, por lo tanto, los problemas a ser resueltos por éste son bastante más sencillos de implementar.

De la misma manera que en el problema anterior, se unificó *Common* con *mGPT* para que la implementación de los problemas pueda ser realizada de la manera que más le convenga al usuario.

Obtención de Política

Dado que una vez ejecutado alguno de los algoritmos de la librería *Common* la política queda guardada en la tabla de hash, este problema pudo resolverse fácilmente al implementar una función que recorre la tabla de hash y obtiene la acción adecuada de acuerdo con la política calculada por el algoritmo.

4.1.2. mGPT

Conexión a servidor

Se añadió una nueva opción *o* que permite al planner trabajar *offline*, es decir, trabajar localmente sin conectarse a un servidor.

Tiempo de algoritmos y heurísticas

Para solucionar este problema fue necesario medir el tiempo que toma el cálculo de cada heurística durante la corrida del algoritmo seleccionado, de esta forma se puede obtener el tiempo exacto que le tomó al algoritmo resolver el problema, sin tomar en cuenta el posible retraso que le haya causado calcular la heurística.

Para las heurísticas cuyo valor se calcula antes de realizar la ejecución del algoritmo sólo es necesario medir el tiempo que tomó calcularla y restarlo al tiempo total cuando el algoritmo finalice. En los casos en los que la heurística se calcula durante la corrida del algoritmo es necesario guardar, también durante la corrida, las pequeñas fracciones de tiempo que tome calcular cada valor de la heurística; para finalmente restarlo al total del tiempo al terminar la ejecución.

Pocos algoritmos

La implementación actual de *mGPT* sólo contiene algunos algoritmos; sin embargo, para completar los objetivos de la investigación se incluyeron también otros tantos tales como: varias versiones de *LRTDP, ILAO* y *LDFS*, los cuales sí están implementados en la librería *Common*. Esto se logró mediante la unificación de ambas librerías.

Como ya se mencionó anteriormente, la unificación también permite utilizar en la librería *Common* las heurísticas más completas de *mGPT*. De esta manera logramos aprovechar al máximo todas la características favorables de las dos librerías que se utilizaron durante la investigación.

4.2. Dominios implementados

4.3. Tron

4.3.1. Descripción del problema

En el juego participan dos jugadores, cada uno de ellos posee una moto que puede mover hacia cualquier dirección (arriba, abajo, izquierda y derecha). Cuando una moto se mueve crea una pared en el lugar donde se encontraba anteriormente, de forma que cada moto va creando una pared más larga a medida que se moviliza en el tablero.

El objetivo del juego consiste en lograr que la moto del jugador oponente se estrelle contra una pared, para ello cada jugador deberá intentar encerrar al oponente con su pared.

4.3.2. Descripción de la Solución

Representación de un estado

Para representar cada estado del juego se creó la estructura **st_tron** la cual contiene los siguientes atributos:

- **Posición de la Moto 1:** Contiene las coordenadas x, y del tablero en las cuales se ubica la moto 1.

- **Posición de la Moto 2:** Contiene las coordenadas x, y del tablero en las cuales se ubica la moto 2.

- **Orientación de las motos:** Para cada estado es necesario saber hacia que dirección apunta cada moto, para ello el estado contiene una variable **dir** de tipo *char*. Los primeros 4 bits indican la orientación de la primera moto y los últimos 4 la orientación de la segunda, donde cada bit representa una orientación (norte, sur, este y oeste) y el bit encendido indica la dirección actual de cada moto.

  Adicionalmente se implementaron funciones para manipular y obtener esta información cuando sea conveniente.

- **Tablero:** Para representar la pista en la que se encuentran las motos, el estado posee una variable de tipo *array* que contiene 2 elementos de tipo *char*. El tablero, de tamaño 4x4, contiene 16 casillas, de esta forma cada bit del arreglo representa una casilla en el tablero.

  Una casilla, en algún momento determinado del juego, puede contener una moto, una pared o estar vacía.

Acciones

En un estado determinado, cada jugador puede moverse una casilla en la dirección que desee (arriba, abajo, izquierda o derecha). Los movimientos que saquen a la moto del tablero no están permitidos.

Dado que cada moto va creando una pared a medida que se mueve tampoco está permitido realizar el movimiento contrario a la orientación de la moto; esto ocasionaría una vuelta de 180 grados que haría que el jugador se estrellara contra la pared que acaba de crear.

En la implementación los movimientos que el jugador puede realizar son independientes de la orientación actual de la moto, es decir, si por ejemplo el jugador decide realizar la acción *arriba* la moto se moverá hacia arriba en el tablero, sin importar la orientación; y se modificará ésta última de acuerdo con la acción.

Estados Terminales

El juego termina si alguna moto se estrella contra una pared que haya sido creada por alguno de los participantes durante el juego, es este caso el jugador sobreviviente es el ganador de la partida.

El juego también termina si ambas motos chocan entre sí, en este caso ninguno de los jugadores es declarado ganador.

Estimaciones de Utilidad Usadas

El costo de realizar una acción es 1. En caso de que la acción lleve a un estado terminal su valor se calcula de la siguiente manera:

- Si ambas motos se estrellan a la vez contra una pared o chocan entre ellas el estado tendrá un costo de 50.
- Si choca la moto controlada por el algoritmo de decisión secuencial el estado tendrá un costo de 1000.
- Si choca la moto cuyo movimiento es no determinístico el estado tendrá un costo de 0. Dado que se busca minimizar los costos, éste es el mejor resultado posible.

4.4. MTS (Moving Target Search)

4.4.1. Descripción del problema

En un laberinto sin ciclos de *NxN* cuadros un depredador debe atrapar a una presa que se mueve de manera no determinística. Tanto el depredador como la presa se mueven un cuadro por vez. Inicialmente el depredador se encuentra en la esquina superior izquierda del laberinto y la presa en la esquina inferior derecha. El objetivo es encontrar una política óptima para que el depredador capture la presa.

El objetivo del experimento es minimizar el costo de llegar hasta la presa. Por cada estado al que llegue el depredador tal que no sea estado terminal se sumará un costo positivo. Un estado terminal tiene costo cero.

4.4.2. Descripción de la Solución

Representación de un Estado

Para representar cada estado del juego se creó la estructura **mts** la cual contiene los siguientes atributos:

- Posición del depredador

- Posición de la presa
- **Solved:** Indicador para identificar los estados terminales.

Representación del Laberinto

La representación de un laberinto se realiza mediante un arreglo de ceros y unos, donde el cero indica que la casilla esta ocupada por una pared, y el uno que se encuentra vacía.

La construcción de un laberinto sin ciclos se hace de la siguiente manera:

- Inicialmente todas las casillas están ocupadas con paredes.
- Utilizando *Depth First Search* se van eliminando las paredes siguiendo los movimientos generados mediante la función *random*.
- Se realizan excavaciones hasta que todas las casillas del laberinto hayan sido visitadas.
- Si se realiza un movimiento y la casilla hacia la que se esta realizando este ya esta vacía el algoritmo se devuelve un paso para evitar el ciclo resultante. De esta manera no se genera ningún ciclo en la ejecución.

Estimaciones de Utilidad Usadas

Por cada estado no terminal que el depredador alcance en su busqueda por la presa se agrega un costo de +1 a la utilidad de la búsqueda. Todos los posibles estados terminales son evaluados de la misma forma con un costo de 0.

4.5. Spy

4.5.1. Descripción del Problema

En un cuarto de tamaño NxM se encuentra un espía y un policía. La habitación está divida en una sección sur y una sección norte. Hay dos pasillos que conectan amabas secciones.

El policía ronda estos dos pasillos cambiando de dirección con cierta probabilidad dada por el usuario. El policía puede detectar al espía dentro de un radio de acción indicado por el usario.

El espía empieza en la esquina inferior izquierda y su objetivo es navegar la habitación hasta alcanzar la puerta de salida en la esquina superior derecha sin que el policía lo detecte.

4.5.2. Descripción de la Solución

Representación de un estado

Para representar cada estado del juego se creó la estructura **spy_t** la cual contiene los siguientes atributos:

- **Posición del policía:** Contiene las coordenadas x, y del tablero en las cuales se encuentra el policía.
- **Dirección del policía:** Contiene la dirección de la ronda del policía, puede ser en sentido horario o anti-horario.
- **Posición del espía:** Contiene las coordenadas x, y del tablero en las cuales se encuentra el espía.

4.5.3. Representación de la Habitación

Para representar la habitación se creó la estructura **room_t**. Esta estructura contiene los siguientes atributos:

- **Dimensión de la habitación:** Contiene el tamaño en las coordernadas x, y de la habitación.
- **Radio:** Tamaño del radio de detección del policía.
- **Probabilidad de cambio:** Probabilidad de que el policía cambie la dirección de su ronda.

- **y1:** Contiene la coordenada y donde comienzan los pasillos.
- **y2:** Contiene la coordenada y donde terminan los pasillos.
- **x1:** Coordenada x del primer pasillo.
- **x2:** Coordenada x del segundo pasillo.

Acciones

En un estado cualquiera el espía puede moverse una casilla en cualquiera de las cuatro direcciones norte, sur, este y oeste; además, puede quedarse en la casilla actual. No es una acción aplicable moverse en una dirección que lleve al espía en contra de una pared.

Estados Terminales

Existen dos tipos de estados terminales, aquellos en los que el espía logró alcanzar la salida y aquellos en los que el policía detectó al espía.

Estimaciones de Utilidad Usadas

Cada acción del espía tiene un costo de 1, bien sea que se haya movido en alguna dirección o que haya decido quedarse.

Si el espía logra alcanzar la salida el costo de este estado es 0. Dado que se busca minimizar los costos, éste es el mejor resultado posible.

Si el policía logra detectar al espía el costo es 1000, con el propósito de penalizar estos estados y de que sean evitados por la política óptima.

Capítulo 5

Experimentos y Resultados

5.1. Dominios

Los dominios utilizados para probar la librería extendida de Common son los que se describieron en la sección de Implementación: Tron, MTS y Spy.

Los dominios utilizados para la evaluación de la fusión de mGPT y Common son: Elevators, BlocksWorld, ZenoTravel, Schedule y Drive. Todos extraidos de la Quinta Competición Internacional de Planificación llevada a cabo en el año 2006.

La definición de estos dominios y sus instancias se obtuvieron como archivo PPDDL de la IPC5.

5.2. Heurísticas Utilizadas

Para los dominios de evaluación de Common se utilizaron las heurísticas *Zero* y *Min-min-value-iteration*. Además, se desarrolló una heurística específica para cada problema: Para MTS se utilizó el costo del camino mínimo divido entre dos; para Tron la distancia mínima desde la moto enemiga a una pared; y para Spy la *Distancia Manhattan* considerando el costo de llegar a los pasillos.

En el caso de los dominios de la IPC5 se utilizaron las siguientes heurísticas: *Zero*, *Min-min-lrtdp* y *Min-min-lrtdp* utilizando a *Atom-min-backwards-1* como heurística base.

Adicionalmente, se desarrolló una heurística general que llamamos *Epsilon-descendiente*. Consiste en utilizar la solución del mismo problema con un epsilon más relajado como heurística para la solución con el epsilon deseado. Pueden realizarse varias iteraciones disminuyendo en cada una el valor de epsilon.

5.3. Experimentos

A continuación se describirán para cada dominio los experimentos que se realizaron.

Todos los experimentos se llevaron a cabo bajo una plataforma Gentoo Linux 2.6, Pentium IV - 2.8/3 GHz, 512MB RAM.

Para todos los experimentos se utilizó un límite máximo de tiempo de 6 horas, un máximo de consumo de memoria real de 512MB y un máximo de consumo de memoria virtual de 512MB.

5.3.1. IPC5

Se ejecutaron corridas con los siguientes parámetros:

Algoritmos: *VI, ILAO**, *LRTDP, LDFS*.

Heurísticas: *zero,min-min-lrtdp,atom-min-backwards-1*.

ϵ: 0.01, 0.001.

Para cada dominio se resolvieron cinco instancias del problema.

5.3.2. Tron

Se ejecutaron corridas con los siguientes parámetros:

Algoritmos: *VI, ILAO**, *LRTDP, LDFS*.

Heurísticas: *zero, min-min-vi, i-manhattan*.

ϵ: 0.01, 0.001, 0.0001, 0.0.

Tamaños del tablero: 4x4, 4x6.

5.3.3. MTS

Se ejecutaron corridas con los siguientes parámetros:

Algoritmos: *VI, ILAO**, *LRTDP, LDFS*.

Heurísticas: *zero, min-min-vi, camino-minimo*.

ϵ: 0.01, 0.001, 0.0001, 0.0.

Tamaños de laberinto: 10, 20, 30, 40, 50.

Número de ciclos: 0, 1, 2, 4, 5.

5.3.4. Spy

Se ejecutaron corridas con los siguientes parámetros:

Algoritmos: *VI, ILAO*, LRTDP, LDFS*.

Heurísticas: *zero, min-min-vi, i-manhattan*.

ϵ: 0.01, 0.001, 0.0001, 0.0.

Tamaños de laberinto: 20, 30, 40, 60

5.3.5. Spy Epsilon-descendiente

Se ejecutaron corridas con los siguientes parámetros:

Algoritmos: *VI, ILAO*, LRTDP, LDFS*.

Heurísticas: ϵ-d 0.01, ϵ-d 1.0|0.01, ϵ-d 0.1|0.001, ϵ-d 0.01|0.001.

ϵ: 0.0001.

Tamaños de laberinto: 30, 40.

5.4. Análisis de Resultados

5.4.1. IPC5

BlocksWorld

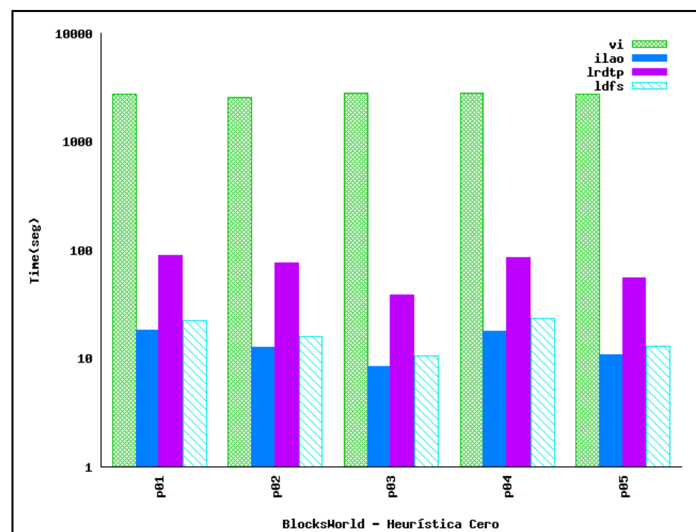

Figura 2: Comparación de algoritmos - heurística 0 - BlocksWorld

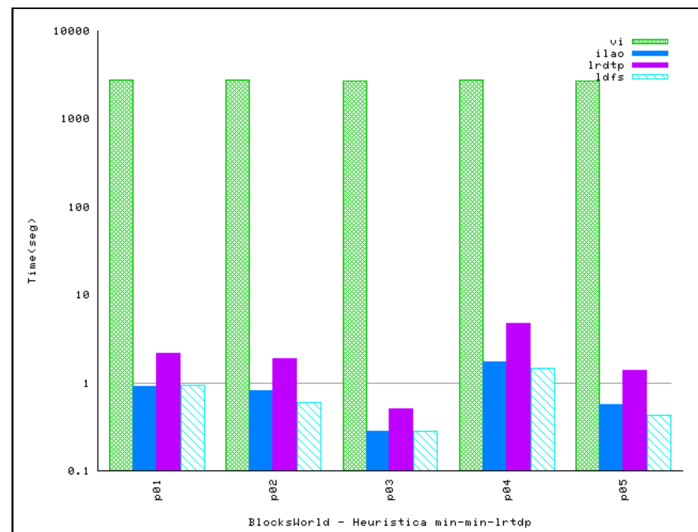

Figura 3: Comparación de algoritmos - heurística min-min-lrtdp - BlocksWorld

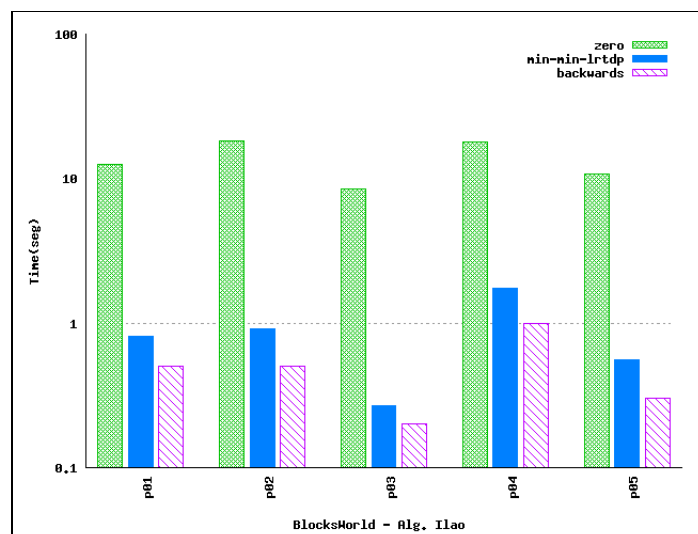

Figura 4: Comparación de heurísticas - BlocksWorld

Los algoritmos presentan un comportamiento bastante uniforme. VI es el menos eficiente de todos siendo superado por el resto en varios órdenes de magnitud. Tanto ILAO* como LDFS son los algoritmos que logran resolver en menor tiempo el problema, el primero aventaja al segundo usando la heurística *Zero*, y viceversa con la heurística *Min-min*. No llega a notarse sin embargo una diferencia notable entre ambos. La heurística *Backwards-1* no se ejecuta dentro de los límites establecidos para una gran cantidad de instancias. El desempeño de LRTDP sí está considerablemente por debajo de los dos an-

teriores aunque sigue siendo mucho mejor que VI. Ver figuras 2 y 3. De las heurísticas es *Atom-min-backwards-1* la que más aporta a la eficiencia de los algoritmos cuando se ejecuta, *min-min-lrtdp* resulta también provechosa evidenciándose esto en que ambas superan a *Zero* ampliamente. Ver figura 4.

Elevators

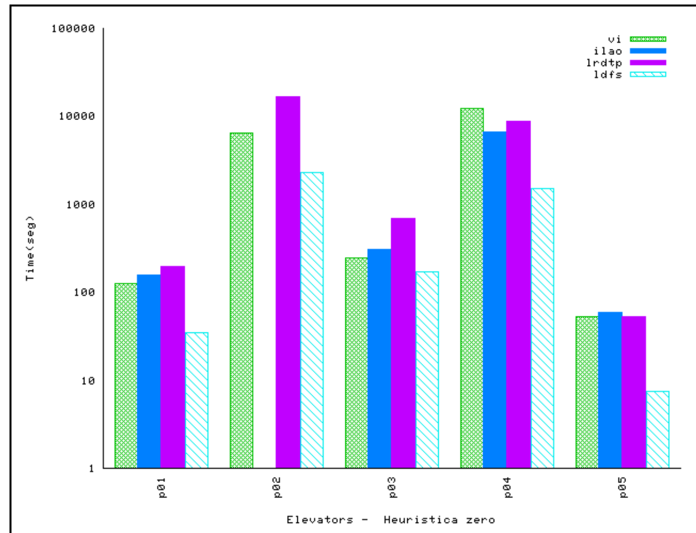

Figura 5: Comparación de algoritmos - heurística 0 - Elevators

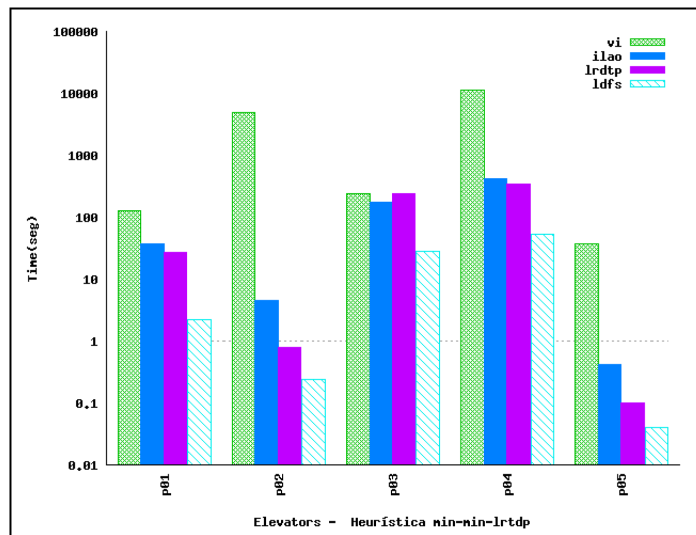

Figura 6: Comparación de algoritmos - heurística min-min-lrtdp - Elevators

LDFS resulta claramente el mejor algoritmo para este dominio. Seguido de LRTDP que

supera a ILAO a excepción de unas pocas instancias. VI es considerablemente más lento que el resto. Ver figuras 5 y 6.

La heurística *Atom-min-backwards-1* no se logró computar en ninguno de los casos por requerir cantidades de memoria que excedían los límites. *Min-min-lrtdp* no presentó este inconveniente y supero claramente a la heurística *Zero*.

Schedule

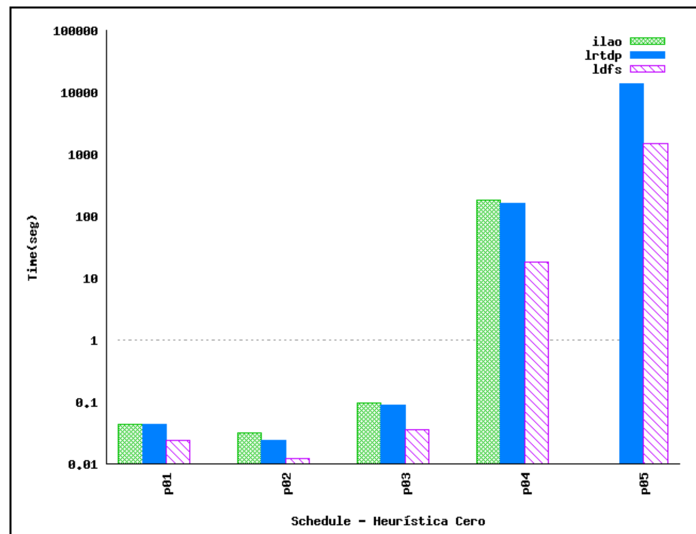

Figura 7: Comparación de algoritmos - heurística 0 - Schedule

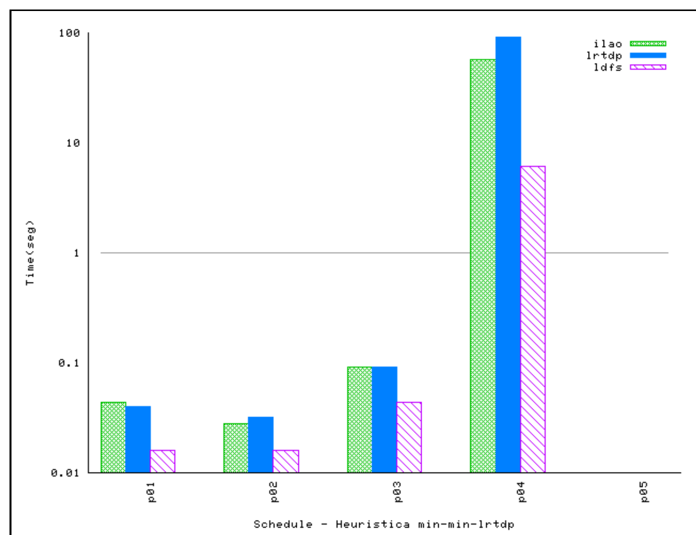

Figura 8: Comparación de algoritmos - heurística min-min-lrtdp - Schedule

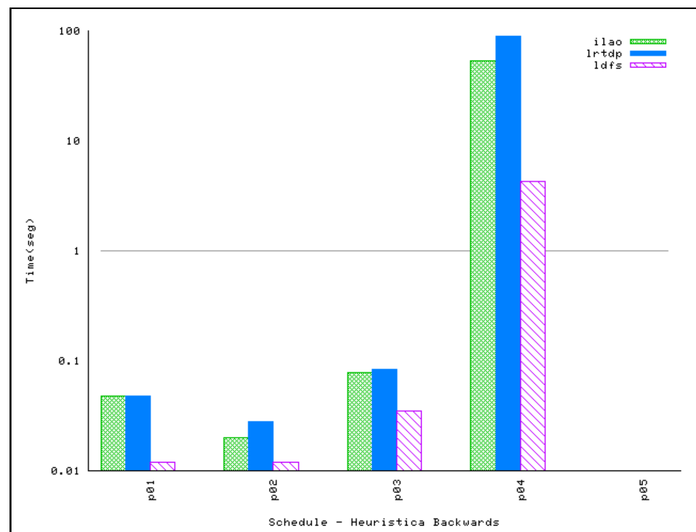

Figura 9: Comparación de algoritmos - heurística Backwards - Schedule

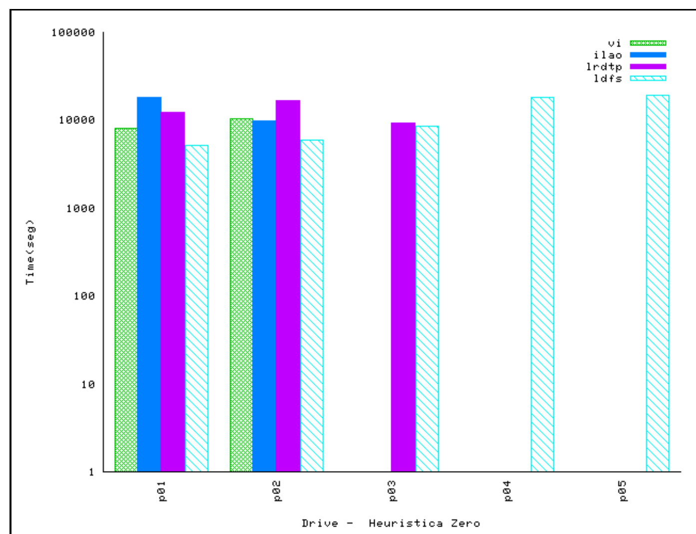

Figura 10: Comparación de algoritmos - heurística 0 - Drive

En este caso se evidencia especialmente el pobre desempeño de VI al no lograr resolver ninguna de las instancias. ILAO y LRTDP tienen más o menos el mismo comportamiento aunque el segundo tiende a superar ligeramente al primero, de hecho, ILAO no es capaz de resolver la instancia más difícil. LDFS es el algoritmo más eficiente en todos los casos y por márgenes a ser tomados en cuenta. Ver figuras 7, 8 y 9.

En relación a las heurísticas, *Atom-min-backwards-1* es una vez más la que mejor resultados logra. Sin embargo, para la instancia más compleja ni ésta, ni *Min-min-lrtdp* logran

computarse por límites de tiempo.

Drive

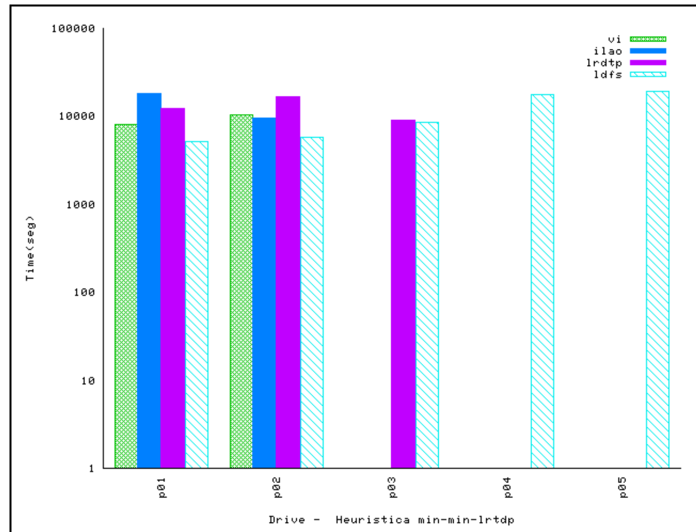

Figura 11: Comparación de algoritmos - heurística min-min-lrtdp - Drive

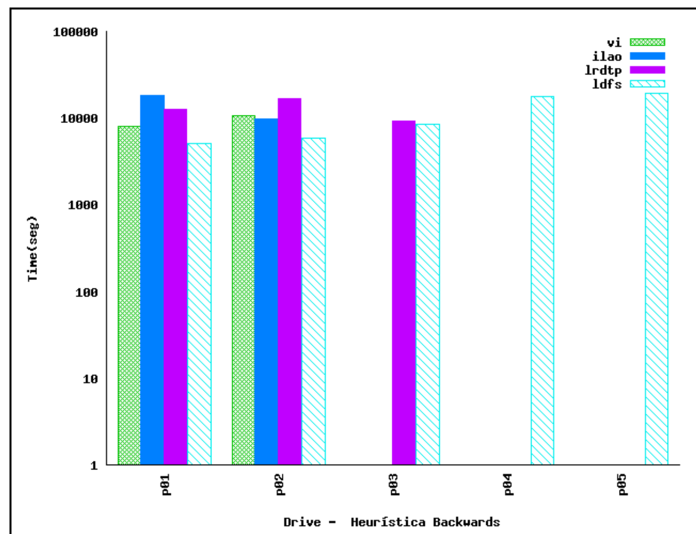

Figura 12: Comparación de algoritmos - heurística Backwards - Drive

En este dominio LDFS es notablemente el mejor algoritmo. Más allá de los tiempos obtenidos la diferencia radica en que logra resolver la totalidad de las instancias mientras sus pares ILAO y VI por la complejidad del problema solamente alcanzan a resolver dos de las cinco. LRTDP por su parte logra encontrar la solución en tres de los casos. Ver

figuras 10, 11 y 12. Ninguna de las heurísticas presenta un comportamiendo destacado y en general los resultados con todas son bastante similares.

Zenotravel

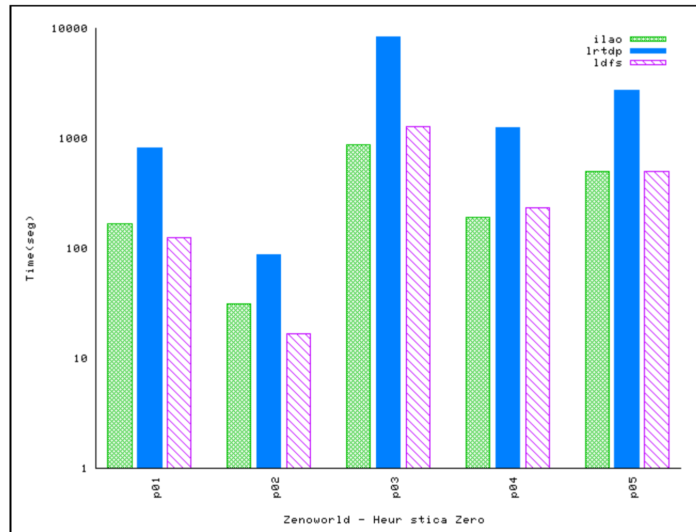

Figura 13: Comparación de algoritmos - heurística 0 - Zeno

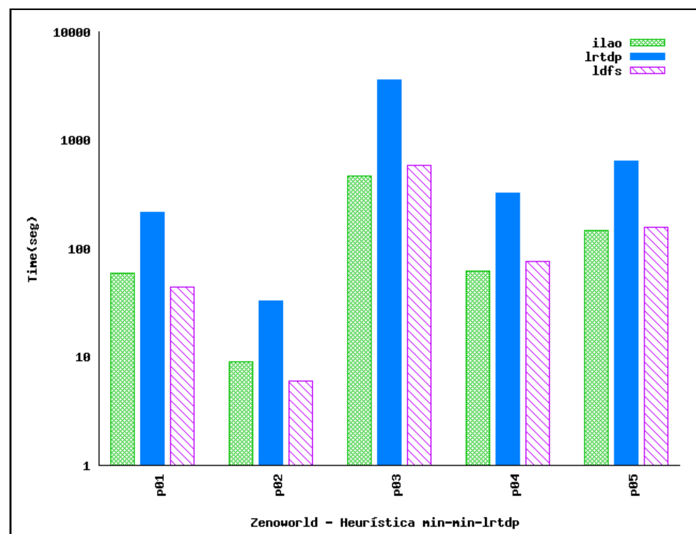

Figura 14: Comparación de algoritmos - heurística min-min-lrtdp - Zeno

No existe un algoritmo claramente superior en este caso, dependiendo de la instancia ILAO o LDFS presentan el mejor desempeño, aunque sin llegar a obtener diferencias notables. Ambos superan claramente a LRTDP. VI una vez más no es capaz de resolver

ninguna de las instancias. Ver figuras 13 y 14. De las heurísticas, *Atom-min-backwards-1* tiene el mejor efecto en los algoritmos en las cinco instancias, seguida de *Min-min-lrtdp*.

5.4.2. Tron

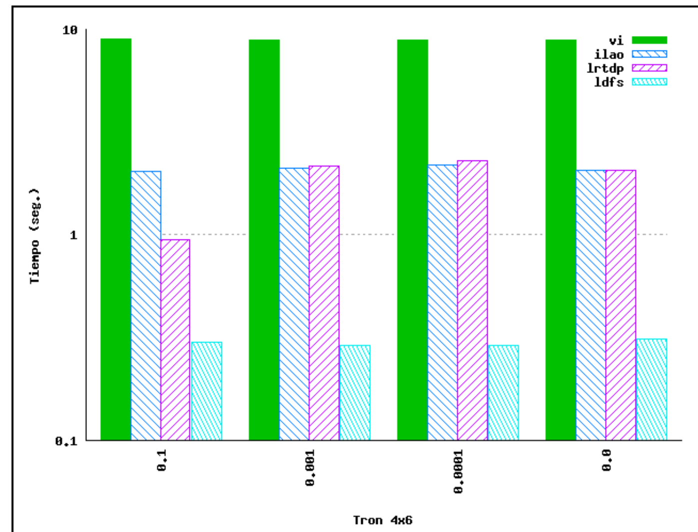

Figura 15: Comparación de algoritmos - heurística 0 - Tron

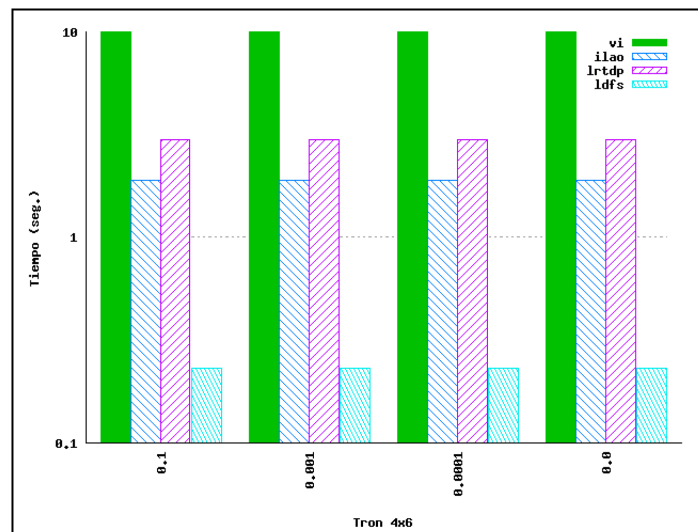

Figura 16: Comparación de algoritmos - heurística min-min-vi - Tron

Para este dominio LDFS supera a los demás algoritmos, con la excepción de la heurística *i-manhattan* para la cual no termina bajo los límites establecidos. Ver figuras 15, 16 y 17

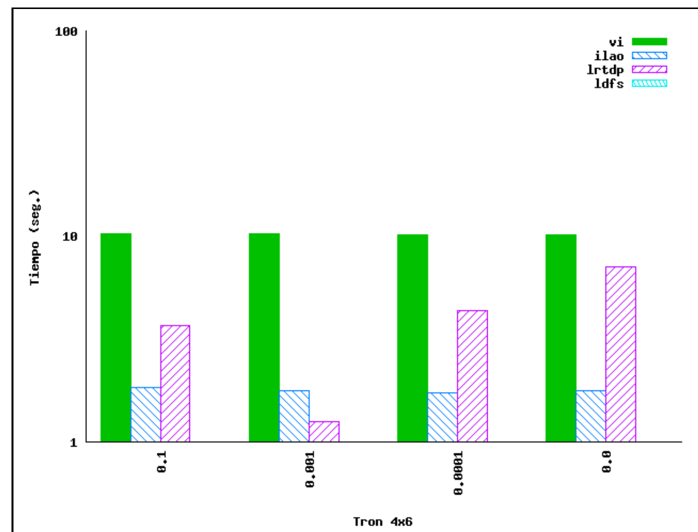

Figura 17: Comparación de algoritmos - heurística dist.min - Tron

En relación a las heurísticas se tiene que el orden de eficiencia es *Min-min-vi*, *i-manhattan* y *Zero* a excepción de algunos casos atípicos en los que la heurística *i-manhattan* se comporta mejor que las restantes.

Con respecto al valor de ϵ se observa que, dado que la cantidad de estados es relativamente pequeña, la dificultad no se incrementa al disminuír este valor. Esto se debe a que el número de estados que es necesario explorar para alcanzar una solución con cierto error no se diferencia de la cantidad necesaria para alcanzar otra solución con un error menor.

5.4.3. MTS

Value Iteration es más rápido que los demás algoritmos en todos los casos. Esto se debe a que en este dominio, el número de estados relevantes en relación al número de estados totales no es significativamente menor. Por lo tanto, algoritmos que se aprovechan de esta característica no pueden hacerlo en este caso.

A diferencia de lo esperado, mientras mayor sea el número de ciclos, el problema no se dificulta.

El orden de velocidad de los algoritmos tiende a ser VI, ILAO*, LRTDP y LDFS. Ver

figuras 18, 19 y 20.

En cuanto a las heurísticas *Min-min-vi* y *Camino-minimo* no se puede establecer el dominio de una sobre la otra.

Mientras más ciclos haya en el laberinto la heurística *Camino-minimo* tarda más tiempo en computarse. Esto se debe a que su implementación está basada en búsqueda en *DFS*.

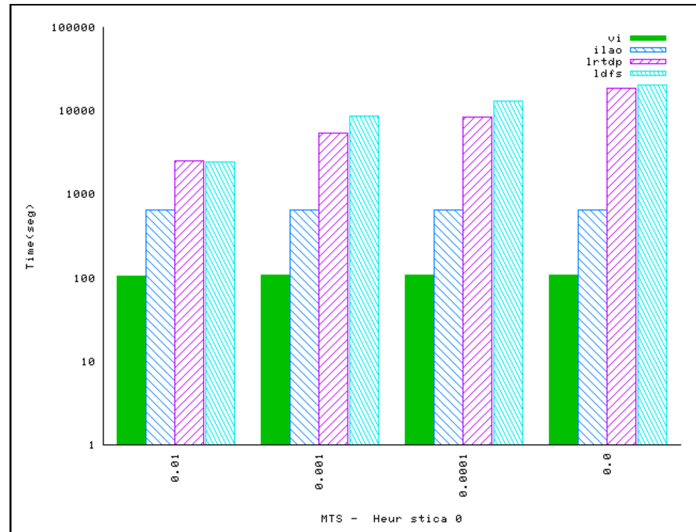

Figura 18: Comparación de algoritmos - heurística 0 - MTS

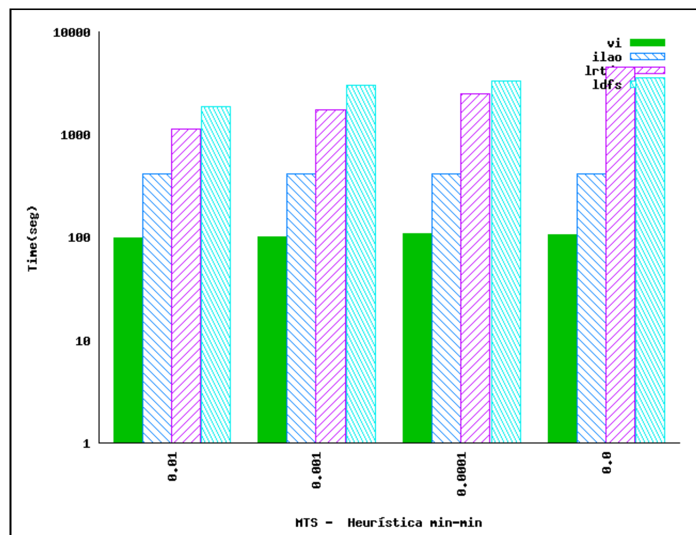

Figura 19: Comparación de algoritmos - heurística min-min-vi - MTS

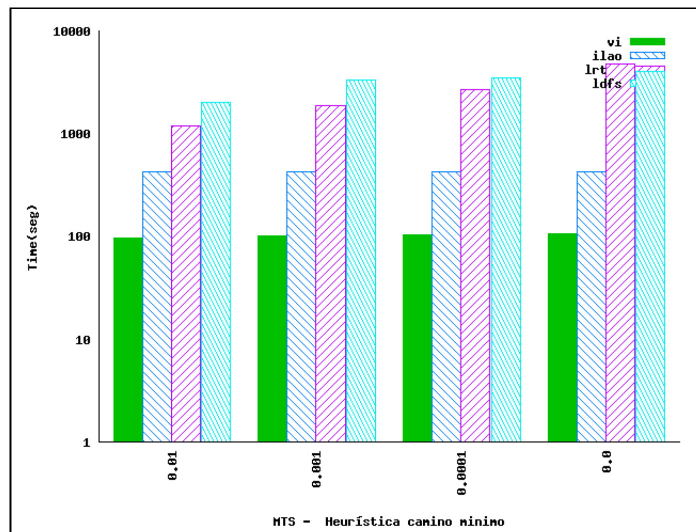

Figura 20: Comparación de algoritmos - heurística cam.min/2 - MTS

5.4.4. Spy

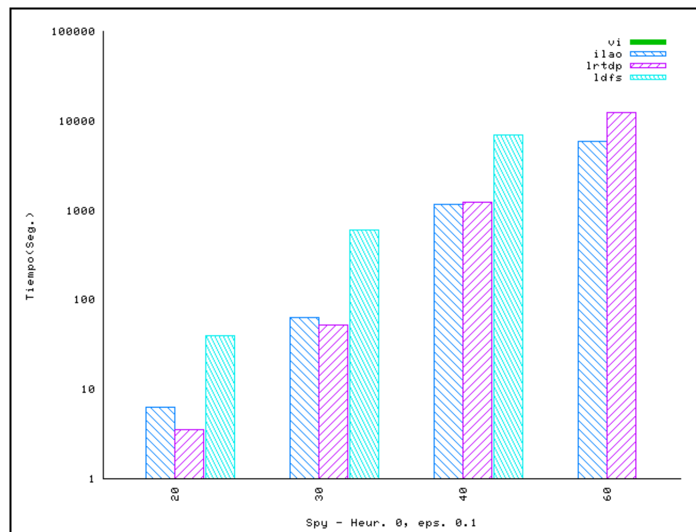

Figura 21: Comparación de algoritmos - heurística 0 - Spy

Este dominio fue diseñado con el propósito de demostrar que los algoritmos *ILAO**, *LRTDP* y *LDFS* se comportan mejor cuando la proporción de estados relevantes versus estados totales es baja. Los estados relevantes en *Spy* son aquellos donde el espía se encuentra en alguno de los pasillos, dirigiéndose a ellos o hacia la salida. Existen una gran cantidad de estados irrelevantes en los que la posición del espía está en una de las dos

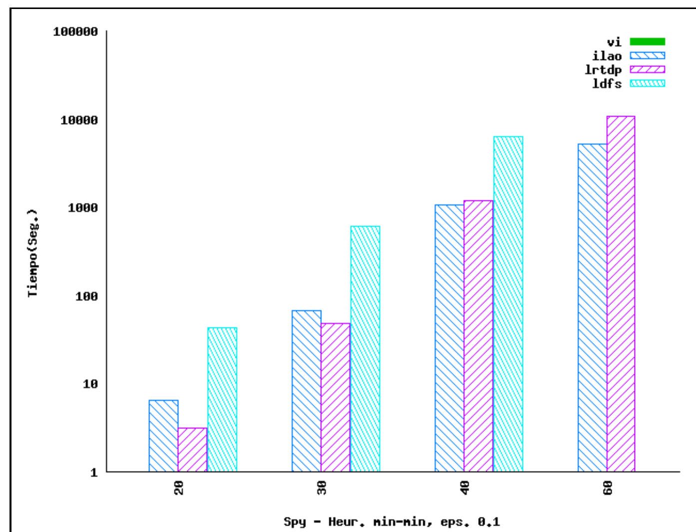

Figura 22: Comparación de algoritmos - heurística min-min-vi - Spy

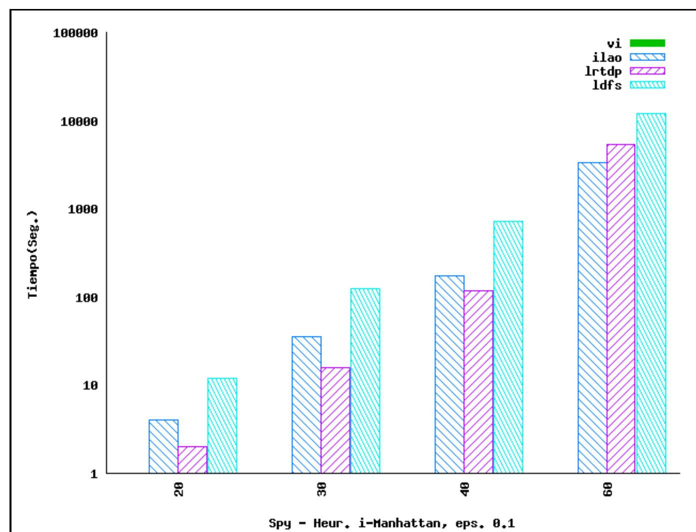

Figura 23: Comparación de algoritmos - heurística i-Manhattan - Spy

esquinas que no son la inicial o la terminal.

Lo primero que se observa es que *Value-Iteration* no logra resolver ninguna instancia de este problema. Ello se debe a que incluso las instancias más pequeñas tienen una gran cantidad de estados.

La razón por la cual los demás algoritmos sí logran converger en el tiempo límite es que la proporción de estados relevantes comparados con la totalidad de los estados es muy pequeña. Estos algoritmos están diseñados para tratar de explorar sólo los estados

relevantes.

En las instancias pequeñas el orden de eficiencia de los algoritmos es LRTDP, ILAO*, LDFS. Ver figuras 21, 22 y 23.

A medida que el tamaño aumenta ILAO* supera en eficiencia a los demás algoritmos.

En todo momento el orden de eficiencia de las heurísticas es *i-manhttan*, *Min-min-vi* y *Zero*. Adicionalmente, el tiempo que consume *i-manhattan* es despreciable y logra mejorar la eficiencia de los algoritmos desde cerca de 50 % hasta un 150 % en algunos casos.

5.4.5. Spy Epsilon-descendiente

Utilizando esta heurística se determinó que la mejor combinación de *epsilon* para *ILAO** es 0,01. En el caso de *LRTDP* y *LDFS* es 0,01|0,001. Ver figura 24.

Adicionalmente, *LRTDP* es el algoritmo que logra resolver los dominios en el menor tiempo utilizando ésta técnica. Ver figura 25.

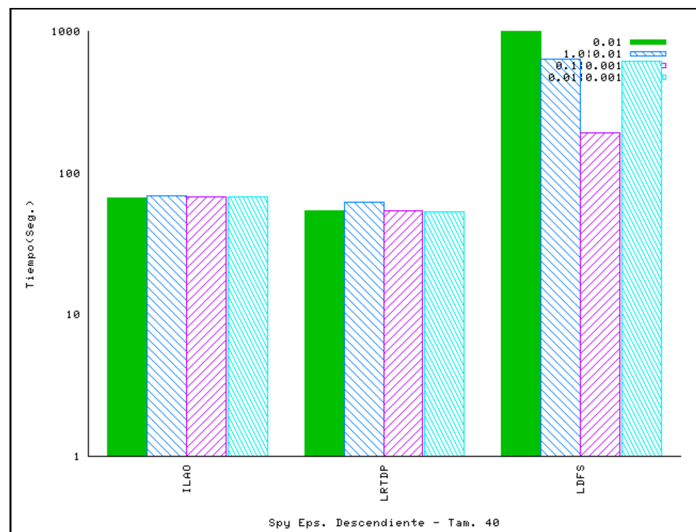

Figura 24: Comparación de heurísticas - Spy ϵ-descendiente

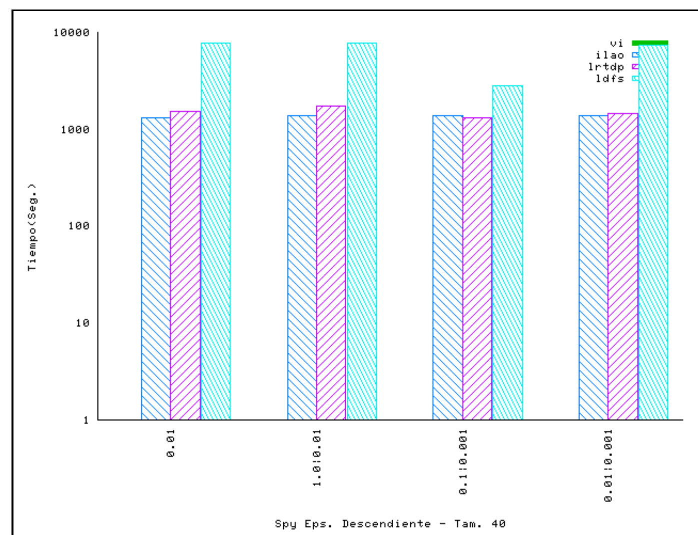

Figura 25: Comparación de algoritmos - Spy ϵ-descendiente

Capítulo 6

Conclusiones y Recomendaciones

6.1. Conclusiones

En general, y más específicamente en los dominios correspondientes al IPC5, LDFS es el algoritmo que logra resolver la mayor cantidad de instancias y a la vez de la manera más eficiente en términos de tiempo. En aquellos casos donde ILAO* presentaba los mejores resultados, LDFS también mostró buen desempeño. Aunque los resultados muestran que LDFS sobresale ante los otros algoritmos, no se puede establecer que alguno sea definitivamente dominante.

Los problemas de IPC5 estaban diseñados para ser resueltos utilizando un modelo de planificación probabilística. Se experimentó con traducirlos a problemas de tipo MDP y luego solucionarlos.

Es importante destacar que las políticas calculadas son óptimas, a diferencia de los resultados alcanzados en la competencia real IPC5, que se basa en un modelo cliente-servidor que permite el cálculo de políticas propias.

Tal como era de esperarse, disminuir el valor de ϵ hace que aumente la dificultad del problema, esto se debe al hecho de que para minimizar el error es necesario realizar un mayor número de *trials* en cada algoritmo.

Del análisis de los resultados de los dominios *MTS* y *Spy* se desprende que la proporción de estados relevantes sobre estados totales influye en el desempeño de los algoritmos. Para *ILAO**, *LRTDP* y *LDFS*, que están diseñados para explorar los caminos más cercanos a la solución mínima del problema, la proporción de estados relevantes debe ser baja con

respecto a la cantidad de estados totales para que funcionen de forma óptima. *Value-Iteration* recorre todo el espacio de búsqueda y por lo tanto se comporta mejor que los anteriores cuando la proporción de estados relevantes es alta.

En cuanto a las heurísticas utilizadas se tiene que la que en general produce el mejor comportamiento es *Atom-min-backwards-1 | Min-min-lrtdp*, se evidencia una reducción drástica en el tiempo de corrida del algoritmo en comparación con la ejecución con otras heurísticas. Sin embargo, requiere muchos recursos para ser calculada, por lo que en instancias difíciles y con nuestras restricciones de espacio y tiempo algunos algoritmos no son capaces de finalizar el cálculo de la política.

Las heurísticas construidas específicamente para los dominios *MTS*, *Spy* y *Tron* utilizan menos tiempo para calcularse, pero no dominan a *Min-min-vi* con la cual se realizó la comparación.

Con respecto a la heurística *ϵ-descendiente* se puede concluir que las combinaciones probadas para *ILAO** y *LRTDP* no son notoriamente mejores que la heurística *zero* o cualquier otra utilizada. Sin embargo, para *LDFS* se comprobó que la combinación 0,1|0,001 resultó ser mucho mejor (hasta 10 veces más eficiente) que todas las demás heurísticas utilizadas en el mismo problema.

Vale la pena resaltar que, para los dominios *MTS* y *Spy*, a medida que el espacio de búsqueda aumenta *ILAO** se comporta mejor que el resto de los algoritmos. En estos problemas *ILAO** escala mejor que los demás.

6.2. Direcciones Futuras

Se recomienda realizar investigaciones más amplias utilizando la heurística *ϵ-descendiente*. Probar más dominios, más instancias y más combinaciones de ϵ en vista de los buenos resultados obtenidos para el algoritmo *LDFS*.

Se descartó uno de los dominios experimentados debido a que presentaba *Dead-Ends* [1] y la librería *Common* no está preparada para manejar estos casos. En vista del buen rendimiento de ésta se recomienda ampliarla para considerar los mismos.

Dado que la fusión *Common + mGPT* arroja soluciones óptimas en tiempos competitivos, se recomienda utilizarlo para futuras ediciones de la IPC.

Muchas instancias y algunos dominios presentaron problemas a causa de limitaciones en nuestros recursos, se recomienda realizar éstos u otros experimentos de la misma clase en estaciones con mayor capacidad de memoria y procesamiento.

Se recomienda realizar una investigación a nivel teórico para determinar la causa por la cual ciertos algoritmos son mejores en determinados dominios.

[1] estados no terminales en los que no hay ninguna acción aplicable

Bibliografía

[Bertsekas, 1995] Bertsekas, D. (1995). Dynamic programming and optimal control. Belmont, MA. Athena Scientific.

[Bonet y Geffner, 2001] Bonet, B. y Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33.

[Bonet y Geffner, 2003] Bonet, B. y Geffner, H. (2003). Labeled RTDP: Improving the convergence of real-time dynamic programming. In Giunchiglia, E., Muscettola, N., y Nau, D., editors, *Proc. 13th International Conf. on Automated Planning and Scheduling*, pages 12–21, Trento, Italy. AAAI Press.

[Bonet y Geffner, 2005a] Bonet, B. y Geffner, H. (2005a). An algorithm better than ao*? In *AAAI*, pages 1343–1348.

[Bonet y Geffner, 2005b] Bonet, B. y Geffner, H. (2005b). mgpt: A probabilistic planner based on heuristic search. In *Journal of Artificial Intelligence Research 24*, pages 933–944. AAAI Press.

[Bonet y Geffner, 2006] Bonet, B. y Geffner, H. (2006). Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to mdps. In *Proc. of 16th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, pages 142–151. AAAI Press.

[Bonet et al., 1997] Bonet, B., Loerincs, G., y Geffner, H. (1997). A robust and fast action selection mechanism for planning. In Kuipers, B. y Webber, B., editors, *Proc. 14th National Conf. on Artificial Intelligence*, pages 714–719, Providence, RI. AAAI Press / MIT Press.

[Fikes y Nilsson, 1971] Fikes, R. y Nilsson, N. (1971). *STRIPS: a new approach to the application of theorem proving to problem solving*, pages 189–208.

[Fox y Long, 2003] Fox, M. y Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. In *Journal of Artificial Intelligence Research 20*, pages 61–124. AAAI Press.

[Hansen y Zilberstein, 2001] Hansen, E. A. y Zilberstein, S. (2001). LAO * : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62. Disponible en: `citeseer.ist.psu.edu/hansen01lao.html`.

[Meyn y Tweedie, 1993] Meyn, S. P. y Tweedie, R. L. (1993). *Markov Chains and Stochastic Stability*. Springer-Verlag, London.

[Rusell y Norvig, 2003] Rusell, S. y Norvig, P. (2003). *Inteligencia Artificial Un Enfoque Moderno*, chapter 17. Pearson Prentice Hall, second edition.

[Younes y Littman, 2004] Younes, K. L. S. y Littman, M. L. (2004). Ppddl1.0: The language for the probabilistic track of ipc-4. Disponible en: `www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/younes.pdf`.