UNIVERSIDAD SIMON BOLIVAR
Decanato de Estudios de Postgrado
Maestría en Ciencia de la Computación

TRABAJO DE GRADO

PLANIFICACIÓN COMO RAMIFICACIÓN Y PODA: UNA IMPLEMENTACIÓN MEDIANTE PROGRAMACIÓN CON RESTRICCIONES

por

Héctor Luis Palacios Verdes

Abril de 2003

UNIVERSIDAD SIMON BOLIVAR

Decanato de Estudios de Postgrado

Maestría en Ciencia de la Computación

PLANIFICACIÓN COMO RAMIFICACIÓN Y PODA: UNA IMPLEMENTACIÓN MEDIANTE PROGRAMACIÓN CON RESTRICCIONES

Trabajo de Grado presentado a la Universidad Simón Bolívar por

Héctor Luis Palacios Verdes

Como requisito parcial para optar al título de

Magister en Ciencia de la Computación

Realizado con la tutoría del Profesor

Carolina Chang

Abril de 2003

UNIVERSIDAD SIMON BOLIVAR
Decanato de Estudios de Postgrado
Maestría en Ciencias de la Computación

PLANIFICACIÓN COMO RAMIFICACIÓN Y PODA: UNA IMPLEMENTACIÓN MEDIANTE PROGRAMACIÓN CON RESTRICCIONES

Esta Trabajo de Grado ha sido aprobada en nombre de la Universidad Simón Bolívar por el siguiente jurado examinador:

Presidente
Prof. Oscar Meza

Miembro Externo
Prof. Alejandro Crema
Universidad Central de Venezuela

Miembro Principal-Tutor
Prof. Carolina Chang

Fecha:_____

Resumen

Dos nociones claves en los problemas de búsqueda heurística y optimización combinatoria, son la ramificación y las cotas inferiores. La búsqueda se realiza pasando de un estado a otro que le sucede. La ramificación se refiere a la manera en que se definen los estados en los cuales puede continuarse la exploración. Las cotas inferiores son medidas aproximadas del costo, usadas para dirigir la elección del siguiente estado, y podar ramas en el espacio de búsqueda que conducirían a falla. En la planificación en Inteligencia Artificial, las heurísticas admisibles o cotas inferiores han recibido una atención considerable, y algunos de los planificadores óptimos actuales las utilizan, de forma explícita o implícita (*e.g.* en un grafo del plan). La ramificación, en cambio, ha recibido menor atención y se discute poco explícitamente. En el presente trabajo, la noción de ramificación en planificación se hace explícita y se relaciona con esquemas de ramificación usados en optimización combinatoria.

Desarrollamos un planificador óptimo paralelo basado en "ramificación y poda" (llamado BBP, por la siglas del nombre en inglés: *Branch and Bound Planner*). Su formulación e implementación permite poner en perspectiva la relación entre diferentes enfoques al formular planificadores, a saber: basados en búsqueda heurística, restricciones, y orden parcial. Éste muestra la relación entre estos enfoques y la complementariedad de muchos de sus planteamientos, lo cual proporciona un enfoque unificado que abre la posibilidad de seguir desarrollando planificadores de mayor rendimiento que puedan resolver problemas de diverso tipo y mayor complejidad. Este trabajo también señala relaciones entre la planificación y otros problemas de Investigación de Operaciones, como la planificación de tareas. En la implementación de BBPse usó programación con restricciones, con la que se representan las cotas inferiores y las decisiones que se toman para realizar la búsqueda. Dicha técnica está siendo utilizada cada vez más para atacar problemas de optimización combinatoria y otros comúnmente resueltos a través de técnicas de Investigación de Operaciones. Al concluir el trabajo se obtuvieron resultados comparables a algunos planificadores de última generación como GraphPlan y HSPr*, sentando las bases para una mayor exploración.

Palabras clave: Inteligencia Artificial, Planificación, Ramificación y Poda, Heurísticas, Programación con Restricciones

Índice general

Índice de figuras

Índice de cuadros

Capítulo I

INTRODUCCIÓN

A fin de enmarcar y motivar este trabajo, mostramos en primer lugar algunos antecedentes del problema de planificación en inteligencia artificial. En particular, nos detenemos en algunas de las soluciones, sobre todo las más recientes. Detrás del éxito de éstas, están presentes varias ideas que además pueden entenderse mejor en el marco de la ramificación y poda. Una alternativa usada actualmente para la implementación de dichas técnicas es la programación con restricciones. Con estos insumos se presenta el trabajo, el cual consiste en un planificador que usa ramificación y poda, e incorporando varias ideas de planificadores recientes. En la formulación e implementación se usa programación con restricciones.

I.1. Antecedentes

Uno de los intereses primarios desde los inicios de la Inteligencia Artificial (IA) en los años 50 fue la resolución de problemas. Se denomina *problema* a un estado inicial, una meta y un conjunto de acciones posibles. Partiendo del estado inicial, se pueden ejecutar varias de esas acciones, hasta llegar a la meta. Este trabajo se enmarca en el objetivo de desarrollar técnicas que permitan construir solucionadores *generales*. Es decir, herramientas donde se puedan especificar problemas en lenguajes de alto nivel y obtener soluciones para ellos.

El problema de la *Planificación*[72] surgió en el contexto de un robot que realiza acciones secuencialmente en un ambiente complejo, partiendo de una situación inicial y tratando de cumplir un objetivo. Es decir, queremos encontrar una secuencia de acciones $a_1, ..., a_n$ que nos lleva de una situación inicial I a una situación meta M; donde cada acción a_i puede ejecutarse si es cierto el conjunto de proposiciones que constituyen su precondición, $prec(a_i)$. En consecuencia, algunas proposiciones serán ciertas en el estado siguiente, $adic(a_i)$, y otras serán falsas, $elim(a_i)$.

Esta manera de representar los problemas fue introducida en STRIPS[28], uno de los primeros sistemas de Planificación. La misma permite expresar situaciones de logística de

transporte de paquetes que incluyan diferentes tipos de vehículos, juegos de cartas, las célebres *torres de Hanoi*, entre otras. Originalmente, la representación no toma en cuenta la duración de las acciones, ni si se pueden realizar varias acciones al mismo tiempo o no (concurrencia); tampoco maneja ningún tipo de incertidumbre tanto en la percepción del ambiente cómo en el efecto de las acciones. Aun siendo este lenguaje muy simplificado, sirvió de base para muchos años de investigación, y hoy en día es usado por la mayoría de los planificadores.

En la actualidad existen principalmente dos enfoques en la *Planificación*[35]:

Planificación independiente del dominio. Se trata de atacar el problema en sí mismo estudiando mecanismos generales de búsqueda, y una representación adecuada del problema a fin de poder realizar la búsqueda más eficientemente. La Planificación Independiente del dominio está relacionada con el interés dentro de la IA por crear herramientas que resuelvan problemas de manera "inteligente" pero sin tener que especificar cómo resolverlos. Entran en esta línea los *Sistemas Expertos* basados en diferentes técnicas, y el uso cada vez más frecuente de lenguajes de alto nivel para solucionar problemas de optimización y búsqueda. En el caso más general, incluye el manejo general de recursos y tiempo, y de las nociones de incertidumbre y no determinismo en su ejecución.

Planificación dependiente del dominio. Trata de definir lenguajes de representación y algoritmos que permitan incorporar en el modelo conocimiento específico que haga factible la solución a problemas de mayor complejidad.

Cuando se hacen aplicaciones industriales, ambas técnicas se utilizan, combinando algoritmos independientes del dominio con información del contexto. También hay una fuerte relación con la *Planificación de Actividades* (en inglés *Scheduling*) y otros problemas de *Investigación de Operaciones*. Las herramientas para atacar todos estos problemas se integran para lograr los resultados que se requieren, manejando apropiadamente la duración de las acciones y los recursos que requieren (ver [2, 87, 63]). En este trabajo se tratará únicamente la Planificación Independiente del Dominio.

I.1.1. Planificación No Lineal

Aún con la simplicidad de la definición de STRIPS, costó desarrollar planificadores que fueran completos para este, es decir, que dieran una solución en tiempo finito siempre que existiera. La solución vino con lo que se conoce como *Planificación de Orden Parcial* [83], llamada así porque se mantiene un orden parcial entre las acciones que deben realizarse.

Este orden permite garantizar que se satisfacen las condiciones necesarias para aplicar cada acción cuando le corresponda. Los planificadores basados en esta técnica generan nuevos planes (incompletos) en cada punto de decisión, por lo que se dice que realizan la búsqueda en el *espacio de planes*. Contrastan con estos los que buscan en el *espacio de estados*, en el que cada decisión es elegir una acción aplicable que generará, a su vez, otro estado.

Este esquema resultó ser muy flexible y permitió grandes avances en la expresividad de los lenguajes, llegando a incluir el manejo de expresiones lógicas más complejas, ejecución condicional y duración de acciones. Sin embargo, los problemas solucionados seguían siendo de tamaño muy reducido.

Estas extensiones al lenguaje que permite expresar problemas de planificación, lo han hecho más cercano a aplicaciones industriales. Incluso algunos planificadores actuales para ambientes complejos usan como base la planificación no lineal. Ejemplos de estos son IxTex[47] y RAX[40].

I.1.2. Avances recientes

Hasta la planificación no lineal, se tendía a trabajar sólo con problemas pequeños a fin de atacar las grandes dificultades teóricas existentes. Pero aún no podían resolverse problemas de complejidad real que pudieran ser usados industrialmente. En ese contexto, a mediados de los noventa se dieron varios intentos por mejorar el rendimiento de los planificadores. Además, el área se volvió mucho más experimental, enfocándose en cómo escalar las soluciones a problemas de mayor envergadura, una vez que se resolvieron ciertos interrogantes teóricos de expresividad de los lenguajes de especificación de problemas, así como las condiciones suficientes para encontrar una solución correcta[52]. Surgieron SATPlan[41], GraphPlan[4], HSP[6] y otros planificadores basados en ideas de estos tres primeros (BlackBox[42], IPP[43], Stan[49], FF[39], GRT[69]).

La mayoría de éstos transforman el problema de planificación en otro problema cuya solución pudiera obtenerse en forma eficiente. SATPlan lleva el problema a un conjunto de fórmulas lógicas que se resuelven con motores de inferencia. GraphPlan genera un grafo que contiene información sobre las acciones aplicables y los conflictos entre ellas; luego busca en ese grafo para generar los planes. HSP transforma el problema en una búsqueda heurística en el espacio de estados, utilizando algoritmos clásicos, pero generando estimaciones del costo (heurísticas) de lograr predicados y acciones a partir de la descripción del problema planteado. Estos planificadores obtuvieron avances significativos en la envergadura de los problemas que se podían resolver. Por ejemplo, todos los problemas que antes servían de referencia para comparar los planificadores, ahora son resueltos

en milésimas de segundos.

Es importante distinguir entre planificadores óptimos (en términos del número mínimo de acciones), y subóptimos. También se encuentran los secuenciales y paralelos (que permiten realizar acciones al mismo tiempo). Se plantea la posibilidad de hacer planificación subóptima porque el problema de planificación, en su versión más simplificada, es PSPACE en complejidad [9] y cualquier sistema se encontrará siempre con la barrera de crecimiento altamente exponencial de su tiempo de respuesta con respecto a la complejidad de los problemas. SATPlan es paralelo, y tiene algoritmos para planificación óptima y subóptima. GraphPlan es óptimo y paralelo. HSP es secuencial y tiene versiones óptimas y subóptimas.

Algunos de estos avances se han mostrado en una competencia que se ha organizado en torno a la conferencia *Sistemas de Planificación en Inteligencia Artificial* (en inglés *Artificial Intelligence Planning Systems*, AIPS[55][1]). A ésta asisten grupos de investigación de todo el mundo y se ha vuelto referencia obligada en la comunidad. Esto ha llevado al establecimiento de un conjunto de problemas que sirven de medida del rendimiento. También permiten una comparación más justa y creíble de los resultados reportados por investigadores independientes.

Otra consecuencia es el uso generalizado de un lenguaje de modelación común en los diferentes planificadores: *Lenguaje de Definición de Dominios de Planificación* (en inglés *Planning Domain Definition Language*, PDDL[54]). Este lenguaje permite modelar problemas en STRIPS e incorpora diferentes extensiones que no todos los planificadores soportan. También se han incorporado al lenguaje las acciones con duración, variables numéricas (versus predicados lógicos) y acciones continuas en el tiempo, entre otras características.

Recientemente los métodos utilizados se acercan a la Planificación de Actividades en el área de Investigación de Operaciones. Ésta y la Planificación en IA pueden entenderse como simplificaciones de un modelo más complejo que incluye duración en las acciones, uso de recursos compartidos y precedencia entre las acciones obtenidas por las condiciones que estas requieren del estado. La integración de la complejidad de cada problema, hace que la solución al problema general sea mucho más dificil. Los resultados actuales son bastante preliminares, pero siendo el modelo mucho más cercano a la realidad, hay gran interes en mejorar el desempeño.

Planificación como Búsqueda Heurística

HSP y toda la familia de planificadores basados en búsqueda heurística (HSP[7][6], FF[39], GRT[69], AltAlt[67]) usan heurísticas que se calculan resolviendo una simpli-

ficación del problema, que permite obtener información sobre los costos reales que ayuda en la toma de decisiones al realizar la búsqueda. Algunas de estas simplificaciones pueden llevar a cotas inferiores (heurísticas admisibles [61]) que permitirían hacer planificación óptima utilizando algoritmos generales de búsqueda como A*[58] o IDA*[44]. En otros casos, los estimados no son cotas inferiores y sólo sirven para obtener soluciones subóptimas. En [36] se da una familia de heurísticas admisibles.

Este enfoque ha sido muy útil en planificación secuencial, sobre todo en dominios que no admiten muchas acciones paralelas. En cambio, en estos últimos dominios tienen más éxito los enfoques basados en satisfacción de restricciones. En una de las últimas competencia AIPS (2000)[1] los planificadores con mejor rendimiento estaban basados o usaban como alternativa la Búsqueda Heurística.

I.2. Algunas hipótesis

De todos estos avances mencionados hay varias ideas que conviene destacar del conjunto. Una es la importancia de calcular cotas inferiores para el costo de lograr ciertos objetivos, lo que permite podar drásticamente el espacio de búsqueda. Lo novedoso es poder sintetizar las cotas a partir de la definición del problema. Dichas heurísticas han sido usadas principalmente en planificadores que buscan en el espacio de estados (familia de planificadores HSP). Cuando los problemas son inherentemente secuenciales, los algoritmos de búsqueda más exitosos son los que mejor escalan (como A*, IDA*[46]), pero tienen problemas para tratar los casos de planificación paralela[36].

En los dominios con paralelismo tiene menos sentido ordenar las acciones a realizar, con lo cual el espacio de búsqueda sobre estados podría crecer artificialmente. Además, podría tener ventajas el tomar la decisión de colocar una acción en cualquier lugar del plan, tal como se hace en los planificadores basados en satisfacción de restricciones o de orden parcial. En este caso se busca en el espacio de planes.

Sin embargo, con algunas excepciones (RePop[59], UnPop[53]), este tipo de planificadores no tienen cotas inferiores que sirvan para reducir el espacio de búsqueda.

Así, la pregunta abierta sobre cómo atacar la planificación paralela óptima usando la búsqueda heurística, que tan buenos resultados ha dado en otros contextos, podría tener respuesta en la búsqueda heurística en el espacio de planes. Por ahora, los mejores planificadores paralelos óptimos pertenecen a las familias GraphPlan y SATPlan.

I.3. Ramificación y poda

En Investigación de Operaciones, la Ramificación y Poda[2] se utiliza para problemas diversos de optimización sujeta a restricciones. La idea general es seleccionar recursivamente una posibilidad entre varias. Si esa posibilidad falla, esa información se utiliza a fin de mejorar el resto la búsqueda.

Los algoritmos utilizados para implementar esta idea suelen ser algoritmos de búsqueda[72] tales como búsqueda en amplitud (BFS, A*[58]) y búsqueda en profundidad (DFS). Es posible usar este enfoque si se encuentra una forma de emplear la información de los intentos fallidos.

Para resolver problemas como el Job-Shop[63], Open-Shop[34] y otros relacionados con la Planificación de Actividades, se utiliza también búsqueda no direccional, utilizando cotas inferiores a fin de reducir el espacio. Las ideas determinantes para obtener resultados en estos problemas son relevantes en la planificación de IA.

I.4. Programación con Restricciones

En los *Problemas de Satisfacción de Restricciones* (PSR) se tiene un conjunto de variables definidas sobre un dominio (discreto o contínuo), y se quiere saber si existe una asignación de variables que satisfaga un conjunto de restricciones. Las soluciones dentro de la IA trataron de obtener soluciones independientes de la semántica específica de las restricciones.

Existe, sin embargo, otro enfoque donde se trata de manejar de manera eficiente cierto tipos de restricciones, y el usuario "programa" la solución al problema que quiere resolver en términos de estas restricciones. Esto último se conoce como Programación con Restricciones[51] y está fuertemente relacionada con la Programación Lógica, teniendo en común el enfoque declarativo y la búsqueda intrínseca que realizan. Se utiliza con frecuencia para resolver problemas de optimización combinatoria[18]. Una manera de implementar la Ramificación y Poda, es usando Programación con Restricciones (PR), donde las desiciones que se van tomando en la búsqueda, así como la información de opciones fallidas, se traducen en restricciones.

En los últimos años ha habido grandes avances en la PR, lo que ha llevado al desarrollo de herramientas comerciales que, usando también algunas técnicas de Investigación de Operaciones, han mostrado utilidad para resolver diferentes problemas en la industria.

Así, la PR podría utilizarse para razonar sobre las restricciones de ordenamiento, calcular heurísticas, y para permitir la utilización de la información obtenida de los intentos

fallidos que se usa en los algoritmos de Ramificación y Poda.

Al momento de explorar el espacio de soluciones, una decisión crucial a tomar es qué variable explorar en primer lugar para acercarse a la solución, y luego qué valores intentar[51]. Es necesario elegir técnicas adecuadas en este sentido, ya que impactan drásticamente en el rendimiento.

I.5. Presentación del trabajo

Se presenta un Planificador Paralelo Óptimo llamado BBP, por el nombre en inglés *Branch and Bound Planner*, en castellano Planificador por Ramificación y Poda. Algunas de las características más importantes es que usa cotas inferiores (heurísticas) para reducir el espacio de búsqueda y guiarla, utiliza la información de los intentos fallidos para la misma reducción, y mantiene la relación entre las acciones y los conflictos en forma parecida a la planificación de orden parcial. El planificador está basado fundamentalmente en el esquema propuesto por Geffner en [31].

El aporte principial de este trabajo es la puesta en práctica de un esquema que integra cálculo de cotas inferiores, como en la planificación basada en búsqueda heurística, y esquemas de ramificación no direccional, como se encuentran en la planificación de orden parcial y las formulaciones basadas en satisfacción de fórmulas lógicas (SAT) o satisfacción de restricciones (CSP).

El problema se plantea en forma de optimización combinatoria, por lo que se decidió utilizar un algoritmo de ramificación y poda, usando programación con restricciones. Esto permite introducir restricciones para hacer las ramas, y propagar la nueva información que se tenga sobre las acciones y la relación entre ellas.

Se obtuvieron resultados preliminares que muestran el potencial de este enfoque para desarrollar planificadores independientes del dominio, de alto rendimiento, que exploten los avances recientes. En particular se alcanzan resultados comparables o mejores a GraphPlan.

A continuación describimos la organización del resto del documento. En el capítulo dos se presenta formalmente el problema de la *planificación*, junto con diferentes enfoques para solucionarlo, destacando la planificación de orden parcial, los basados en búsqueda heurística, y los paralelos basados en SAT o CSP. Se indican algunas extensiones que tiene relación con el problema que nos ocupa. Resaltamos las lecciones aprendidas en los diferentes enfoques, apuntando al esquema que finalmente se propone en un capítulo posterior.

En el capítulo tres, introducimos algunos problemas de *optimización combinatoria* que tienen relación con la planificación. Una opción actual para atacar estos problemas es la *programación con restricciones*, que se presenta en el capítulo cuatro.

En el capítulo cinco damos el esquema de ramificación y poda que se usa en la implementación. El planificador BBP se reporta en el capítulo seis, partiendo desde un algoritmo básico, y enriqueciendolo con técnicas usadas en otros planificadores, en la programación con restricciones y la optimización combinatoria.

En los capítulos siguientes se evalua el rendimiento en problemas utilizados por la comunidad, comparándolo con otros planificadores del estado del arte. Se discuten los resultados, llegando a conclusiones y enunciando trabajos futuros que pueden realizarse.

Capítulo II

PLANIFICACIÓN

Como señalabamos anteriormente, queremos desarrollar Solucionadores Generales de Problemas. En el caso particular de la Planificación queremos tener lenguajes generales para especificar cierto tipo de problemas, y dar algoritmos para solucionar estos. Así, en este capítulo presentaremos el problema de *Planificación* mediante un lenguaje específico de representación. Partiremos desde la definición clásica, y llegaremos a mencionar algunas extensiones recientes, en particular la planificación paralela y temporal.

No hablaremos de la aproximación al problema, el enfoque "basado en casos", que tiene que ver con la utilización de planes ya existentes para resolver tareas actuales. Este enfoque es complementario (ver [32]) a las técnicas que presentan un plan nuevo, que son las estudiadas en este capítulo.

Luego veremos los algoritmos más resaltantes que se han dado al problema, resaltando los últimos avances, en cuyo contexto se desarrolla este trabajo. Destacaremos algunas lecciones aprendidas los últimos años, y algunas conjeturas o ideas por explorar.

II.1. Planificación: definición clásica y extensiones

II.1.1. Definición del problema

En la definición clásica de la *Planificación* en Inteligencia Artificial queremos encontrar una secuencia de acciones $a_1, ..., a_n$ que nos lleva de un situación inicial I a una situación meta M. Dichas situaciones están descritas como un conjunto finito de predicados P, definidos sobre un dominio que supondremos también finito[1] $D = d_1, ..., d_m$. Llamaremos P al conjunto de todos los *átomos* o predicados instanciados posibles.

Cada una de las acciones $a \in A$ constan de:

[1]La definición que estamos dando podría ser más expresiva si levantamos esta suposición. Pero la comunidad ha tendido a simplificar el problema así.

- Precondiciones (*Prec*): conjunción de átomos $\{p_i\}$. Cuando son ciertos, la acción es aplicable.
- Adiciones (*Adic*): átomos $\{c_i\}$ que pasan a ser ciertos en el estado resultante de aplicar la acción.
- Eliminaciones (*Elim*): átomos $\{f_i\}$ que pasan a ser falsos en el estado resultante de aplicar la acción.

En principio, las acciones pueden contener en su descripción predicados sobre variables, lo que se conoce como *esquemas* de acciones. Siendo que el domino lo estamos suponiendo finito, podemos tener en vez de cada esquema, todas las posibles instanciaciones completas de las variables del esquema por objetos del dominio. En adelante supondremos que todas las acciones están instanciadas.

De manera que diremos que un *problema* de *planificación* es la tupla $\langle P, I, M, A\rangle$, donde P es el conjunto de todos los átomos posibles, I es la situación inicial, M es la situación meta, y A es el conjunto de todas las acciones instanciados posibles.

El lenguaje para representar las acciones, con estados como conjuntos de predicados y esquemas de acciones, se conoce como STRIPS[28]. Éste vino a ser una solución que manejaba adecuadamente la expresión de los cambios entre los estados, sin tener que referirse extensivamente a hechos que permanecen sin cambios. Aún hoy esta manera de describir los problemas sigue siendo soportado por los planificadores más recientes.

En este trabajo, al instanciar las variables en todos los objetos posibles, obtenemos STRIPS proposicional. Podrían tenerse algunas extensiones iniciales, como precondiciones negativas, metas con literales negativos, y algunas otras[62]. Se ha mostrado que estas extensiones son equivalentes en expresividad a STRIPS proposicional, aunque tengan ventajas desde el punto de vista de la facilidad para modelar problemas, al ser más expresivas. En cuanto a la instanciación de los predicados, la mayoría de los planificadores actuales exitosos en STRIPS, asumen esta restricción.

II.1.2. Modelo matemático asociado

En el problema que definimos, estamos suponiendo en principio que las acciones tienen un efecto determinístico: que se conoce exactamente la única situación a la que lleva el aplicar una acción en una situación específica, de ser aplicable. Y que, además, conocemos con exactitud la situación de la que partimos. Para modelar matemáticamente ésto, es útil un modelo de estado que consta de lo siguiente:

- Un espacio de estados E discreto

- Un estado inicial $e_0 \in E$
- Un conjunto de estados meta $E_g \subseteq E$
- Acciones aplicables $A(e)$ para cada estado
- Una función de transición entre estados: $e' = next(a, e), a \in A(e)$
- Un costo $c(a, e) > 0$ asociado a cada acción en cada estado

Una solución óptima en este modelo es una secuencia de acciones $a_1, ..., a_n$, tal que $e_1 = next(a_1, e_0) \wedge e_2 = next(a_2, e_1) \wedge ... \wedge e_n = next(a_n, e_{n-1}) \wedge e_n \in E_g$. Y $\sum_{i=1}^{n} c(a_i)$ sea mínima.

Siendo así, podemos establecer una asociación de un problema de planificación con un *modelo de estado*, como vemos en la figura II.1.

- Los estados e son conjuntos de átomos (predicados instanciados)
- Un estado inicial $e_0 = I$
- Los estados meta $e_m \in E_g$ son todos aquellos para los que $M \subseteq e_m$. Es decir, tienen que estar todos los atomos de M, pero podría haber otros.
- Acciones aplicables $A(e) = \{a \mid a \in A \wedge Prec(a) \subseteq e)$
- La función de transición entre estados. El estado resultante de aplicar una acción o es: $e' = (e - Elim(o)) \cup Adic(o)$
- $c(o, e) = 1$ asociado a cada acción en cada estado

Figura II.1: Modelo de estados del problema de planificación

Obsérvese que estamos asumiendo que el mundo es *cerrado*. Es decir, que todos los átomos que no son explícitamente ciertos en un estado, por pertenecer al conjunto, entonces los consideramos falsos.

Una ventaja de la formulación STRIPS es permitir capturar cierta expresividad de la lógica de primer orden, manteniendo la tratabilidad de lenguaje. En particular es importante poder capturar en forma sencilla la relación entre los literales y las acciones. Por décadas se ha estudiado el problema, llegando a la utilización de lenguajes generales para especificar acciones, tales como el cálculo de situaciones y el cálculo de eventos[72]. Sin embargo, aún dada la sencillez de las tareas iniciales planteadas, ayudaron en poco a expresar el problema adecuadamente. La implementación de sistemas usando lenguajes teóricamente más complejos que STRIPS tiene hasta el día de hoy serios problemas de escalabilidad.

El éxito de STRIPS muestra también la importancia de tener lenguajes lo suficientemente generales como para tener expresividad al modelar problemas, pero que permitan desarrollar algoritmos eficientes. En la siguiente sección veremos las diferentes soluciones que se han dado a la planificación y cómo ellas se valen de esta representación en particular.

II.1.3. Lenguaje de Definición de Dominios de Planificación (PDDL)

En el marco de la Competencia de Planificación (*Planning Competition* [55][1]), realizada desde 1998 en torno a la conferencia Sistemas de Planificación en Inteligencia Artificia (*Artificial Intelligence Planning Systems*, AIPS), se ha utilizado un lenguaje particular para representar los diferentes dominios, caracterizados por un conjunto de predicados y esquemas de acciones; y los problemas que definen a su vez los estados iniciales y finales. Este lenguaje se conoce como Lenguaje de Definición de Dominios de Planificación (**P***lanning* **D***omain* **D***efinition* **L***anguaje*, PDDL[54][29]).

En su última versión incluye diferentes extensiones a STRIPS, con variadas opciones de complejidad en precondiciones, efectos y metas, y manejo de cantidades, tiempo y recursos.

En vez de desarrollar la sintaxis y semántica del lenguaje, optaremos por dar algunos ejemplos de diferente complejidad, que presenten la versión mas simple del lenguaje, y los diferentes problemas que pueden expresarse en el formalismo. Esto también permitirá captar la expresividad de STRIPS, que es subconjunto de la de PDDL.

Un ejemplo ampliamente usado es el de los *Bloques*. Este consiste en unos bloques dispuestos en pilas sobre una mesa que son manipulados por los brazos de un robot. Los predicados para describir esa situación son:

- `(on ?x ?y)`: El bloque ?x está encima del bloque ?y.
- `(on-table ?x)`: El bloque ?x esta sobre la mesa.
- `(clear ?x)`: No hay ningún bloque sobre ?x.
- `(arm-empty)`: El brazo del robot está vacío.
- `(holding ?x)`: El brazo del robot está sosteniendo el bloque ?x.

.

En la figura II.2 se indican los estados iniciales y finales de un problema de planificación. En PDDL éste quedaría como:

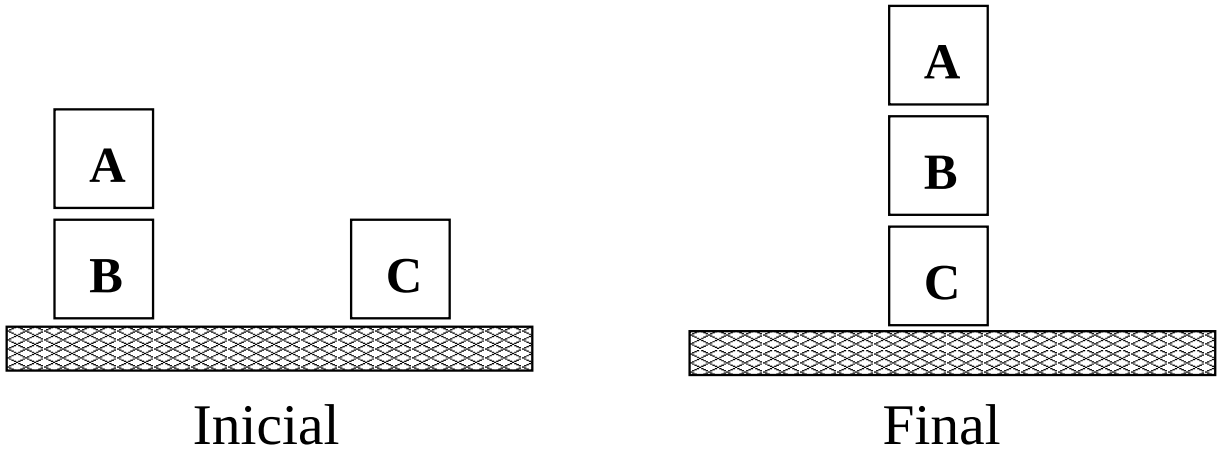

Figura II.2: Estado inicial y final para un problema de Bloques

```
(define (problem problema-simple)
  (:domain blocks)
  (:objects A B C)
  (:init (arm-empty)
         (on-table B)
         (on A B)
         (clear A)
         (on-table C) (clear C))
  (:goal (and
           (clear A)
           (on A B)
           (on B C))))
```

Nótese como el estado inicial esta completamente especificado, al decir que el brazo del robot esta inicialmente vacío. Pero en el estado final no se pide nada con respecto al estado del robot.

Se especifica que los objetos válidos son `A`, `B` y `C`, y que el dominio a usar es `blocks`. Este incluye la descripción de las acciones, en términos de los predicados:

- `(pick-up ?x)`: Toma `?x` con el brazo desde la mesa.
- `(put-down ?x)`: Suelta `?x` en la mesa.
- `(stack ?x ?y)`: Coloca `?x` encima de `?y`.
- `(unstack ?x ?y)`: Toma `?x` de encima de `?y`.

El dominio `blocks` esta definido a continuación:

```
(define (domain blocks)
  (:requirements :strips)
  (:predicates (on ?x ?y)
               (on-table ?x)
               (clear ?x)
               (arm-empty)
               (holding ?x))
  (:action pick-up
     :parameters (?ob1)
     :precondition (and (clear ?ob1) (on-table ?ob1) (arm-empty))
     :effect
        (and (not (on-table ?ob1))
             (not (clear ?ob1))
             (not (arm-empty))
             (holding ?ob1)))
  (:action put-down
     :parameters (?ob)
     :precondition (holding ?ob)
     :effect
        (and (not (holding ?ob))
             (clear ?ob)
             (arm-empty)
             (on-table ?ob)))
  (:action stack
     :parameters (?ob ?underob)
     :precondition (and (holding ?ob) (clear ?underob))
     :effect
        (and (not (holding ?ob))
             (not (clear ?underob))
             (clear ?ob)
             (arm-empty)
             (on ?ob ?underob)))
  (:action unstack
      :parameters (?ob ?underob)
      :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
      :effect
         (and (holding ?ob)
              (clear ?underob)
              (not (clear ?ob))
              (not (arm-empty))
              (not (on ?ob ?underob)))))
```

Por ejemplo, la acción `stack`, coloca el objeto que está en el brazo del robot (`?ob`) sobre otro objeto (`?underob`). Requiere que esté tomando el objeto –`(holding ?ob)`– y que el otro donde lo va a colocar esté libre –`(clear ?underob)`–. El efecto es que ya no

está tomando el objeto, ni esta libre `?underob`. En cambio, el objeto colocado está libre –`(clear ?ob)`–, fue colocado –`(on ?ob ?underob)`– y el brazo del robot quedó libre: `(arm-empty)`. `unstack` toma con el brazo un objeto que estaba encima de otro. `pick-up` y `pick-down`, toman o sueltan, respectivamente, objetos desde la mesa.

En resumen, luego de definir los predicados, se dan los cuatro esquemas de acciones disponibles: poner y quitar, de la mesa o encima de otro bloque. Están indicadas las precondiciones. En los efectos, los predicados que aparecen en forma positiva corresponden a las *adiciones* y los que aparecen negados, a las *eliminaciones*.

Las soluciones a este tipo de problema tienen una alta interacción entre las acciones, ya que un bloque que se cambie de lugar en el momento inadecuado puede acabar con la correctitud de una solución.

Pueden también representarse problemas de búsqueda clásicos como las Torres de Hanoi, con una sola acción que mueve el disco `?disc` desde el disco `?from` al `?to`:

```
(define (domain hanoi)
  (:requirements :strips)
  (:predicates (clear ?x)
               (on ?x ?y)
               (smaller ?x ?y))

  (:action move
      :parameters (?disc ?from ?to)
      :precondition
         (and (smaller ?to ?disc)
              (on ?disc ?from)
              (clear ?disc)
              (clear ?to))
      :effect
         (and (clear ?from)
              (on ?disc ?to)
              (not (on ?disc ?from))
              (not (clear ?to)))))
```

Pero hay dominios con mucha menor interacción entre las diferentes acciones. Por ejemplo, el dominio `Logística` (en inglés *logistics*) corresponde a un problema de transporte dentro de ciudades, y entre ellas, mediante camiones y aviones, respectivamente. Los objetos (`OBJ`) pueden ser transportados en camiones (`TRUCK`) que se mueven dentro de las localidades (`LOCATION`) de una misma ciudad (`CITY`); Los aviones (`AIRPLANE`), en cambio, sólo se mueven entre los aeropuertos (`AIRPORT`) de las diferentes ciudades. Hay acciones para cargar y descargar camiones (`LOAD-TRUCK`, `UNLOAD-TRUCK`) y aviones (`LOAD-AIRPLANE`,

UNLOAD-AIRPLANE), para mover camiones (DRIVE-TRUCK) y aviones (FLY-AIRPLANE). El dominio esta definido a continuación:

```
(define (domain logistics)
  (:requirements :strips)
  (:predicates (OBJ ?obj)
               (TRUCK ?truck)
               (LOCATION ?loc)
               (AIRPLANE ?airplane)
               (CITY ?city)
               (AIRPORT ?airport)
               (at ?obj ?loc)
               (in ?obj ?obj)
               (in-city ?loc ?city))

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

  (:action LOAD-AIRPLANE
      :parameters (?obj ?airplane ?loc)
      :precondition
         (and (OBJ ?obj)
              (AIRPLANE ?airplane)
              (LOCATION ?loc)
              (at ?obj ?loc)
              (at ?airplane ?loc))
      :effect
         (and (not (at ?obj ?loc))
              (in ?obj ?airplane)))

  (:action UNLOAD-TRUCK
      :parameters (?obj ?truck ?loc)
      :precondition
         (and (OBJ ?obj)
              (TRUCK ?truck)
```

```
            (LOCATION ?loc)
            (at ?truck ?loc)
            (in ?obj ?truck))
    :effect
       (and (not (in ?obj ?truck))
            (at ?obj ?loc)))

(:action UNLOAD-AIRPLANE
    :parameters (?obj ?airplane ?loc)
    :precondition
       (and (OBJ ?obj)
            (AIRPLANE ?airplane)
            (LOCATION ?loc)
            (in ?obj ?airplane)
            (at ?airplane ?loc))
    :effect
       (and (not (in ?obj ?airplane))
            (at ?obj ?loc)))

(:action DRIVE-TRUCK
    :parameters (?truck ?loc-from ?loc-to ?city)
    :precondition
       (and (TRUCK ?truck)
            (LOCATION ?loc-from)
            (LOCATION ?loc-to)
            (CITY ?city)
            (at ?truck ?loc-from)
            (in-city ?loc-from ?city)
            (in-city ?loc-to ?city))
    :effect
       (and (not (at ?truck ?loc-from))
            (at ?truck ?loc-to)))

(:action FLY-AIRPLANE
    :parameters (?airplane ?loc-from ?loc-to)
    :precondition
        (and (AIRPLANE ?airplane)
             (AIRPORT ?loc-from)
             (AIRPORT ?loc-to)
             (at ?airplane ?loc-from))
    :effect
        (and (not (at ?airplane ?loc-from))
             (at ?airplane ?loc-to))))
```

No se toman en cuenta tiempos de traslado, costos de cada acción, uso de combustible,

entre otras cosas. Pero este dominio en particular, junto con los bloques han sido campo importante de exploración y comparación de resultados.

II.1.4. Extensiones

Tal como está definido el problema, tenemos varias suposiciones explícitas e implícitas que conviene enumerar, a fin de explorar diferentes complicaciones.

A continuación enumeramos una serie de *suposiciones*, y comentamos el efecto que puede tener el levantar dicha suposición.

Sup1 *Sólo conjunciones positivas en precondiciones.*
Es apenas el comienzo de las posibles extensiones para acercar la expresividad de STRIPS a la lógica de primer orden sin llegar a la complejidad de esta, que impida hacer inferencia en forma eficiente. Destaca en este camino el lenguaje ADL[62] que extiende STRIPS para acercarse más al cálculo de situaciones. PDDL incluye opciones para manejar esta extensión. También hay otras alternativas que utilizan más que predicados para representar precondiciones y efectos[30].

Sup2 *Necesidad de tener una secuencia de acciones.*
Si permitimos que dos o más acciones se ejecuten simultáneamente, debemos evitar que interfieran entre sí, puesto que no está definido el efecto que tiene modificar átomos que formen parte de la precondición de una acción en ejecución, ni el efecto de eliminar un átomo que se esté añadiendo.

Sup3 *Costos iguales y misma duración para todas las acciones.*
Esto tiene que ver con el criterio que se tiene para considerar un plan mejor que otro. El criterio clásico es preferir un plan más corto. Podrían definirse costos diferentes para diferentes acciones, y optimizar sobre el costo total del plan. O tener diferentes duraciones para cada acción. Esto último combinado con el permitir acciones concurrentes, lleva a lo que se conoce como *Planificación Temporal*, que veremos más adelante (sección II.5).

Sup5 *Los efectos de las acciones son hacer cierto o falso un átomo.*
Otra posibilidad es tener cantidades numéricas que varíen con el tiempo, para representar posiciones de objetos y otros escalares.

Sup6 *Ausencia de Recursos.*
La noción de recursos limitados para poder ejecutar ciertas acciones es de gran importancia en la *Investigación de Operaciones*. Si se consideran de manera general,

pueden verse como cantidades que van variando en el tiempo (ver **Sup5**). Un caso particular interesante es cuando existe sólo un elemento del recurso, de manera que sólo puede ejecutarse a la vez una de las acciones que tienen necesidad del recurso.

Sup7 *Conocimiento exacto del estado inicial y del efecto de las acciones .*
Si levantamos esta suposición tenemos *Planificación bajo Incertidumbre*. Algunas soluciones no manejan explícitamente la incertidumbre, sino que dan planes que reducen esa incertidumbre hasta llegar al objetivo con certeza [78][17]. Otros tienen nociones explícitas de la incertidumbre, acciones de observación del ambiente, manejando distribuciones de probabilidad del estado en que están, y efecto probabilístico de las acciones [25][75][5].

Sup9 *Necesidad de optimalidad de la solución.*
El problema de encontrar un plan óptimo para la versión de STRIPS que presentamos es PSPACE-COMPLETO[9]. De allí que encontrar soluciones aproximadas es tremendamente pertinente. En esta tarea ha habido importantes avances los últimos años, como veremos en la siguiente sección.

Queda ahora por ver qué soluciones se han planteado en *Planificación clásica* y algunas extensiones que nos interesan particularmente.

Nótese que el uso de un modelo de estado está asociado a que tenemos información completa y determinismo de las acciones. Ante la falta de determinismo, un modelo apropiado y ampliamente usado son los procesos de Markov (*Markov Desición Process*[66]), y si se tiene información incompleta, con operaciones para observar el ambiente, un modelo podría ser un Proceso de Markov Parcialmente Observable (*Partially Observably Markov Desición Process*[56, 85]).

II.2. Soluciones

En esta sección veremos diferentes soluciones que se han dado el problema que acabamos de presentar. En primer lugar la Planificación de Orden Parcial, que vino a ser *la* solución durante largo tiempo. Luego presentaremos varios avances de la última década.

II.2.1. Planificación de Orden Parcial (POP)

A pesar que puede plantearse naturalmente una solución al problema buscando en el espacio de estados, esta solución pura dista mucho de escalar apropiadamente a solucionar problemas de mediano tamaño. Desde los comienzos de la planificación, la mayoría de

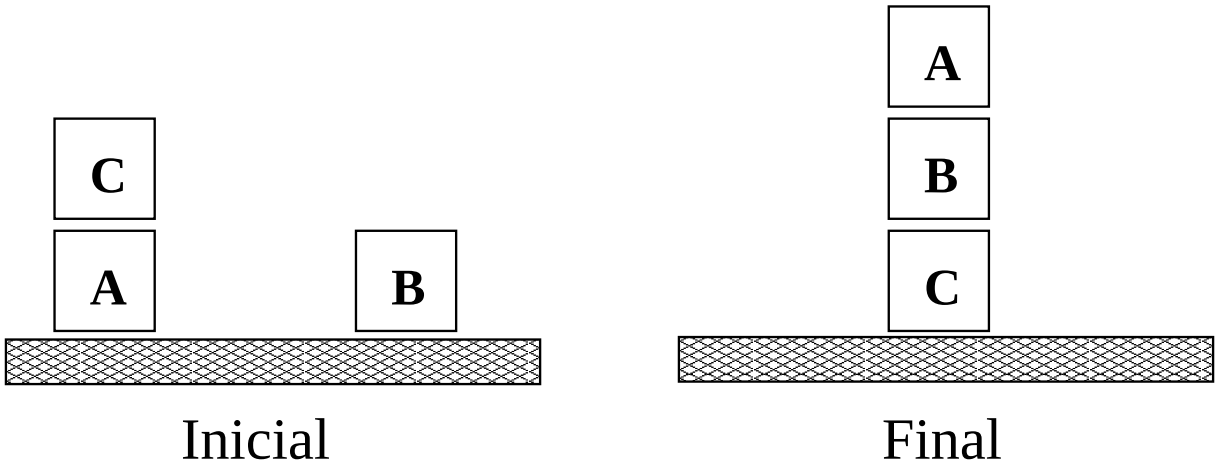

Figura II.3: Estado inicial y final para la Anomalía de Sussman

las soluciones utilizaban el paradigma *divide y vencerás*, para ir resolviendo cada una de las metas, e ir construyendo incrementalmente una secuencia ordenada de acciones. Pero esta estrategia tenía la seria desventaja de no manejar apropiadamente las interacciones al lograr diferentes sub-metas. Por ejemplo, el problema en la figura II.3 (tomada de [72], p. 365) se ilustra un famoso caso llamado la *Anomalía de Sussman* (*Sussman anomaly*), que siendo muy sencillo, los planificadores iniciales fallaban en resolver. El problema consiste en que al descomponer el objetivo, y tratar de colocar el bloque **A** sobre el bloque **B**, las acciones destruyen el haber colocado el bloque **B** sobre el **C**. Los algoritmos lineales no disponen de mecanismos para resolver estas interacciones.

La solución vino con los planificadores no lineales que comienzan a aparecer a mediados de los años setenta[73]. A estos también se les conoce como *Planificadores de Orden Parcial* (POP), puesto que van introduciendo precedencias entre las acciones, sin llegar necesariamente a ordenarlas en forma total. Chapman [15] obtiene resultados formales de correctitud y completitud para los algoritmos de POP. Mc Allester y Rosenblitt [52] introducen el concepto de *Enlace causal*, que resulta clave e ilustraremos más adelante. En [72] puede encontrar una explicación *in extenso* de los algoritmos, comentarios de los algoritmos y enfoques en la *Planificación* hasta antes de la llegada de GraphPlan, que veremos en la sección II.4.1.

Los algoritmos de orden parcial, son aún hoy de los más expresivos para resolver el problema de planificación. Con ellos se ataca casos más complejos desde el punto de vista lógico: precondiciones o efectos cuantificados universal o existencialmente, literales negados en las precondiciones. Son también extensibles a manejo de duración de las acciones y uso de recursos. De hecho, algunos de los planificadores actuales más flexibles utilizan

POP, cómo IxTex[47] y RAX[40] que trabajan en ambientes complejos como misiones espaciales.

Queremos mantener un orden parcial de las acciones que forman parte del plan que nos llevará a lograr el objetivo; manteniendo también los objetivos por resolver, para los cuales aún no se prevé una manera de lograrlos. El algoritmo avanza en refinamientos sucesivos, bien sea ordenando dos acciones a fin de que no interfieran entre sí y anulen efectos de alguna de ellas que son necesarios para lograr el objetivo final; o resolviendo uno de los objetivos p para el cual aún no se posee una solución adecuada, eligiendo una acción a de manera que esta *provea* el átomo: $p \in Adic(a)$. Si en algún momento se falla en la toma de estas decisiones, se vuelve atrás y se intenta otra opción. De esta manera se está realizando una búsqueda en el espacio de los planes parciales, buscando un plan parcial que no tenga átomos para los cuales haya que buscar quien los provee, ni exista ningún conflicto potencial.

Nótese que estamos hablando de un orden parcial de las acciones. Esto puede entenderse como un conjunto de acciones, con relaciones de orden. Pero esto implica que cada acción sólo podría aparecer una vez en el plan. En contraposición, en el modelo de estados que vimos en la figura II.1 se describía la solución como una secuencia de acciones que, por supuesto, admite repeticiones. No todos los problemas pueden resolverse con planes sin repeticiones de acciones. De manera que supondremos que las acciones en POP pueden contener repeticiones de una misma acción, y que las relaciones de precedencia se refieren también a estas acciones distinguiendo las diferentes repeticiones. Para enfatizar esta distinción, en este contexto hablaremos de *Pasos*, para referirnos a la instancia de una acción.

En resumen, entenderemos por *Plan*, en el contexto de POP, a una tupla $\langle$ *Pasos, Orden, Vínculos* $\rangle$. Donde *Pasos* son las pasos que forman parte del plan. *Orden* es el conjunto de relaciones de orden, $paso_i \prec paso_j$, que constituyen el orden parcial entre los pasos del plan. *Vínculos* es un conjunto de estructuras de la forma $P_i \xrightarrow{a} P_j$, que indica que el paso P_i añade el átomo a que es precondición del paso P_j.

Comenzamos inicialmente con un plan con sólo dos pasos: $Inicial$, $Final$. Donde $Inicial$ es un paso introducido artificialmente, que no tiene precondiciones, añade todos los átomos que son ciertos en el estado inicial, y elimina los que son falsos en el estado inicial. $Final$ no tiene efectos, y tiene como precondición los átomos que forman parte del estado final. Este plan inicial no tiene ninguna relación de precedencia. Diremos que un plan es *completo* si al dar un orden total consistente con el orden parcial inducido por las relaciones de precedencias, este plan es correcto en términos del espacio de estados definido en la figura II.1. El plan inicial no es completo. Lo que se hace es tratar de completarlo

eligiendo algo a *reparar* del plan. Esto puede ser:

* Encontrar un paso para proveer una precondición que necesite ser soportada. Este paso podría estar o no en el plan. Todas los pasos del plan necesitan, a su vez, que cada átomo de su precondición tenga un soporte.

* Resolver un *ataque* a un enlace causal. Esto es que un paso P podría eliminar un átomo a del enlace $P_i \xrightarrow{a} P_j$. En este caso uno podría elegir entre *ascenderlo* ($P_j < P$) o *descenderlo* ($P < P_i$). En ambos casos se garantiza que P no borra el átomo a en el contexto de que es necesario para soportar la acción P_j.

Resumiendo, un algoritmo de POP puede observarse en la figura II.4 (basada en [83]). Una característica de este es tener la búsqueda "dirigida por la meta", *i.e.* que trata de colocar los pasos en función de las metas que se quieren obtener al final. Este esquema es representativo de la familia de algoritmos que en torno a la POP se hicieron durante más de una década. En éste se hace una búsqueda en el *espacio de planes*, puesto que cada nodo de la búsqueda es un plan, hasta que se llega a un plan completo. Esta aproximación dió mejores resultados que la búsqueda en el espacio de estados, aun en versiones donde se buscaba desde la meta hacia atrás, que tenían también la propiedad de ser "dirigida por la meta". Esta conclusión sería, sin embargo, revertida más tarde en el contexto de planificación secuencial, con avances que hicieron de la búsqueda una aproximación factible al problema.

Si quisiera hacerse planificación óptima con POP, bastaría con imponer una cota a la distancia entre la acción inicial y la final en el grafo inducido por las relaciones de precedencias. E ir relajando esta cota hasta encontrar la solución.

Algunas lecciones que conviene destacar de todo este esfuerzo son:

- La noción de *enlace causal*, que permite verificar eficientemente si un orden parcial de acciones en efecto llega a la meta.

- La *flexibilidad* del esquema que permite extenderlo con extensiones a STRIPS, duración de acciones, manejos de recursos.

- La idea de no tener que decidir que acción tomar, sólo en un estado. Uno puede hacer compromisos en cualquier lugar del plan, con lo cual al tener más opciones que tomar, puede usarse mejor la información disponible para comparar estas diferentes opciones.

POP($\langle Pasos, Orden, Vinculos\rangle$, $PorSoportar$, $Enlaces$)

1. terminar si $PorSoportar = \emptyset$
2. se selecciona y extrae $\langle Atomo, Paso\rangle \in PorSoportar$
3. **elegir** paso P para soportar $Atomo$:

   - se **elige** un paso que ya está en el plan: $P \in Pasos$.
   - se **elige** un paso nuevo P, aunque pueda coincidir con uno que ya este en el plan.
     Se agrega $Inicial < P$, y a $PorSoportar$ se le agrega $\{\langle Atomo, P\rangle / Atomo \in Prec(P)\}$, *i.e.* los átomos que son necesarios soportar ahora que esta P en el plan.

   en ambos casos, se agrega el enlace causal $P \xrightarrow{Atomo} Paso$ y la precedencia $P < Paso$.
4. Para cada paso P_a que pueda estar atacando un enlace $P_i \xrightarrow{a} P_j$, **elegir** entre:

   - $P_a < P_i$
   - $P_j < P_a$

5. llamar recursivamente a POP con parámetros actualizados.
6. si alguna **elección** falla, tomar otra opción. Sino queda, volver con falla a la elección anterior.

Llamar inicialmente a POP con un plan que consta de los pasos $Inicial$ y $Final$, donde $Inicial < Final$. Y todas las precondiciones de $Final$ (los átomos meta) están en $PorSoportar$. $Enlace = \emptyset$

Figura II.4: Algoritmo de Planificación de Orden Parcial.

Sin embargo, la velocidad de soluciones no escaló apropiadamente principalmente debido a que siendo esto una búsqueda, como la que se pudiera hacer en el espacio de estados, está planteada como una búsqueda *no informada*. Es decir, sin información heurística [61] que permitiera enfocar la búsqueda en las opciones que tengan más posibilidad. Más adelante se verá como esta noción fue fundamental para escalar el rendimiento en planificación secuencial.

II.2.2. Avances recientes

En la década pasada hubo importantes avances que han superado por órdenes de magnitud el rendimiento de los algoritmos POP. Entre ellos están planificadores como SATPlan[41] (óptimo o sub-óptimo paralelo), GraphPlan[4] (óptimo paralelo), HSP[6] (óptimo o sub-óptimo secuencial) y otros basados en ideas de estos planificadores (BlackBox[42], IPP[43], Stan[49], FF[39], GRT[69]). La mayoría de éstos llevaba el problema de planificación a otro problema cuya solución pudiera obtenerse en forma eficiente.

En torno a estos avances se ha organizado la *Competencia de Planificación* (*Planning Competition*) de *AIPS*, donde grupos de investigación de todo el mundo compiten y que se ha vuelto referencia obligada en la comunidad. Esto ha llevado al establecimiento de un conjunto de problemas que sirvan de medida al rendimiento. También permite la comparación más justa y creíble de los resultados reportados por investigadores independientes. Parte de este éxito se debe al uso de PDDL como lenguaje de modelación común a los diferentes planificadores. En las secciones siguiente estudiaremos estos avances y las lecciones que de allí se puede extraer. Comenzaremos por *Planificación como Búsqueda Heurística* puesto que actualmente se presenta como la mejor estrategia en el dominio de planificación secuencial. Más adelante, veremos los diferentes enfoques en planificación paralela. Entre ambos problemas hay fuertes relaciones y diferencias que tomaremos en cuenta al momento de discutir los resultados y lanzar nuevas direcciones.

II.3. Planificación como Búsqueda Heurística

Cómo indicábamos, el problema de planificacíon puede ser resuelto por búsqueda en el espacio de estados que definimos en la figura II.1, utilizando algoritmos de búsqueda en grafos, como la *búsqueda en amplitud*[19]. El problema es que estos no escalan apropiadamente por lo que en inteligencia artificial se habla de algoritmos de búsqueda informados, como A*[58] o IDA*[44]. Para poder usar estos algoritmos es necesario tener una función heurística[61] que permita evaluar cuán lejos esta un nodo con respecto a la meta. Pero en el caso de la planificación tendríamos que tener una función heurística para cada problema STRIPS que se presente, lo cual no concuerda con nuestra intención de hacer planificación independiente del dominio. La *Planificación como Búsqueda Heurística* se volvió factible desde que se introdujo la idea de sintetizar la heurística a partir de la especificación en STRIPS del problema, resolviendo una simplificación de este para obtener información sobre los costos reales que ayude en la toma de decisiones al realizar la búsqueda.

Los primeros trabajos en este sentido fueron UnPop[53] y ASP[8], donde se estimaba el costo de llegar a la meta desde cada estado en particular. Basado en algunas ideas de ASP

está HSP[7][6] y toda una familia de planificadores posteriores (GRT[69], AltAlt[67], FF[39]). HSP transforma el problema a una búsqueda heurística en el espacio de estados utilizando algoritmos clásicos, pero generando estimaciones (heurísticas) del costo de lograr predicados y acciones a partir de la descripción del problema planteado. Entre las diferentes formas de síntesis utilizadas algunas generan funciones heurísticas que permiten hacer planificación óptima, y en otros casos sub-óptimas (planes no necesariamente óptimos). Este enfoque ha sido muy útil en planificación secuencial, sobre todo en dominios que no admiten muchas acciones paralelas. En estos tienen más éxito los enfoques de satisfacción de restricciones que veremos más adelante. De hecho, en la competencia de planificación en el marco de AIPS (2000)[1] los planificadores con mejor rendimiento estaban basados o usaban como alternativa la Búsqueda Heurística.

En adelante presentaremos en detalle a HSP, varias de las formas planteadas para la síntesis de las heurísticas, y su uso en búsqueda direccional (hacia adelante o hacia atrás) en planificación secuencial. Estas ideas también se han usado en planificación paralela, pero ese tema lo veremos más adelante. Para ello repasaremos las heurísticas y algunas propiedades importantes sobre ellas, a fin de poner las bases para mirar en perspectiva las diferentes versiones que se tienen, y las motivaciones y retos detrás de los esfuerzos por mejorarlas.

II.3.1. Búsqueda informada y Heurísticas

En la búsqueda en amplitud[19] se logra obtener el camino óptimo para llegar desde un estado inicial a un nodo que cumple con cierta condición. Esto se hace explorando progresivamente en primer lugar las zonas que, de encontrar la solución, darían un menor costo. Al proceder de esa manera se garantiza que nunca se obviará una solución mejor; de manera que al encontrarla se garantiza su optimalidad. Pero si tuviera un estimado (heurística) de cuan lejos esta la meta desde cada estado, podría usarse este estimado, junto con el costo acumulado de llegar a cada estado, para ordenar de manera más informada dichos estados. Podría explorar el espacio más eficientemente y encontrar una solución con mayor velocidad, sin tener que perder necesariamente la optimalidad de la solución. Esta es la idea detrás de algoritmos como A*[58] o IDA*[44].

Una *función heurística* (simplemente *heurística*) en IA es un estimado del costo de llegar a una meta a partir de un estado. Llamaremos $h^*(e)$ a la función que retorna el costo exacto para llegar desde e a la meta. Si $h(e)$ es una heurística, podría utilizarse $f(e) = g(e) + h(e)$, donde $g(e)$ es el costo de llegar desde el estado inicial, para ordenar los estados eligiendo primero aquellos con menor $f(e)$. Si se hace de manera similar a una búsqueda en amplitud, el algoritmo resultante se conoce como A* [58]. Para obtener

soluciones óptimas la heurística tiene que ser *admisible*, esto es, que no sobre-estime el costo real de llegar a la meta. Un ejemplo de heurística admisibles es la distancia euclidiana, en el contexto del problema del agente viajero[48]. El recorrido que implica llegar de una ciudad a otra por las vías que estipula el grafo nunca podrá ser menor que la distancia física que hay entre dichos puntos. Es común utilizar soluciones simplificadas de los problemas para generar heurísticas.

Si $h(e)$ es admisible, A* garantiza encontrar una solución óptima[61]. Al igual que la búsqueda en amplitud, es exponencial en tiempo y uso de espacio. Por esto último para espacios grandes se vuelve infactible. Una alternativa ampliamente usada es IDA*[44] que usa menos memoria, siendo lineal en la profundidad de la búsqueda. Este realiza una búsqueda en profundidad acotada por una profundidad máxima. Se elige de entre los hijos de cada estado que se acaba de expandir, el que tenga menor $f(e)$. Suponiendo por simplicidad que los costos de las acciones son naturales, comenzariamos desde 1. Si no se encuentra solución a profundidad n, quiere decir que la solución se encuentra al menos a profundidad $n+1$, con lo que se inicia una búsqueda a ese nivel. De esta manera se logra encontrar una solución óptima.

IDA* tiene el mismo orden de comportamiento en tiempo de A*, aunque en la práctica es más lento puesto que tiene que volver a explorar repetidamente muchas regiones. En espacios de estados inmensos, como el que nos ocupa, algoritmos como este son la única posibilidad. En [46] Korf resuelve instancias de *24-puzzle*, que es un rompecabezas donde se tienen 24 piezas más un espacio en blanco, en un tablero de 5 por 5. Sólo se pueden mover las piezas adyacentes al espacio en blanco, y se parte desde un estado inicial, hasta llegar a uno "ordenado" pre-establecido. Esta es una variante del mucho más conocido *8-puzzle*. Puede encontrarse una revisión mas detallada sobre algoritmos de búsqueda, informados o no, en [72], y una discusión profunda sobre las heurísticas en [61].

II.3.2. Heurísticas y Algoritmos

Para hacer planificación por medio de búsqueda heurística, habría que decidir que algoritmos utilizar y que función heurística. Comenzando por esta última, partiremos del hecho conocido de que muchas veces se utilizan versiones más simples del problema a resolver que puedan ser calculadas rápidamente, y que permitan obtener un estimado de cuan lejos está el estado de la meta.

Como primera aproximación consideraremos ignorar las eliminaciones para todas las acciones de un problema P, con lo que obtenemos un problema P'. Podría mostrarse que el costo óptimo $h'(e)$ en P' es inferior al costo óptimo $h^*(e)$, asociada al problema P. Pero resolver P' para obtener $h'(e)$ es NP-completo. Por lo que se plantea otra aproximación.

Supongamos ahora que las sub-metas son *independientes*. Es decir, que lograr un conjunto de metas es equivalente a lograr cada una de ellas por separado, sin tomar en cuenta que lograr una puede entorpecer o colaborar en el logro de la otra. Así, para calcular el costo de una meta descrita por un conjunto de átomos, sólo hay que sumar el costo de lograr cada átomo. El costo de un átomo p es la manera menos costosa de lograrlo, esto es, el costo de lograr las precondiciones de la acción menos costosa para lograr p, más el costo de dicha acción, asumiendo costo cero para los átomos que estén en el estado inicial.

Así, llamando $g_e(p)$ al costo de lograr el átomo p desde el estado e, podríamos definir el estimado como:

$$g_e(p) = \begin{cases} 0 & \text{if } p \in e \\ \text{mín}_{a \in O(p)}[c(a) + g_e(Prec(a))] & \text{en otro caso} \end{cases} \tag{II.1}$$

Donde $O(p)$ es el conjunto de acciones que añaden p, es decir, $O(p) = \{a \mid p \in Adic(a)\}$. El costo $g_e^+(C)$ de un conjunto C de átomos, partiendo desde el estado e puede expresarse como:

$$g_e^+(C) = \sum_{q \in C} g_e(q) \quad \text{(costo aditivo)} \tag{II.2}$$

Recordando que M son los átomos meta, la definición de la heurística h_+ queda:

$$h_+(s) \stackrel{\text{def}}{=} g_s^+(M) \tag{II.3}$$

Esta función estima el costo de alcanzar los atomos en M a partir del estado s, es decir, estima la distancia desde el estado s hasta la meta. Para calcular algorítmicamente h_+ partimos inicialmente de:

$$\begin{aligned} g_s^+(p) &= 0 \quad p \in e_0 \\ g_s^+(p) &= \infty \quad \text{en otro caso} \end{aligned}$$

y luego, usando programación dinámica, hacemos la siguiente actualización hasta alcanzar punto fijo:

$$g_s^+(p) := \text{mín } [\ g_s(p)\ ,\ c(a) + g_s^+(Prec(a))\ ] \tag{II.4}$$

Esta forma de cálculo está relacionada con los algoritmos para encontrar la ruta más corta (*shortest path*[16]) en un grafo donde los nodos son los átomos, partiendo desde los que están en el estado inicial (e_0). Una diferencia con las versiones normales de estos algoritmos es que hay que hacer un ajuste para manejar la minimización que se hace sobre

los subconjuntos de átomos. En general todas las heurísticas que veremos en esta sección se calculan de manera similar: programación dinámica, algoritmos de ruta más corta o, menos comúnmente, *programación con restricciones*.

h_+ no es admisible, puesto que al ignorar la posible colaboración al lograr diferentes sub-metas, podría estar sobre-estimando el costo. Sin embargo, fue utilizada con gran éxito en planificación secuencial sub-óptima, principalmente a raíz de un trabajo llevado adelante por Bonet y Geffner, HSP[7], quienes utilizan esta heurística junto con una versión modificada de A* conocida como WA*. Ésta utiliza la función $f(e) = g(e) + w * h(e)$, donde w es un numero pequeño mayor que 1, que permite llegar más rápido a la meta, aunque sacrificando optimalidad. HSP participó en las competencias de planificación de 1998 y 2000 con tal éxito que en este momento se considera esta *la manera* de afrontar la planificación secuencial.

En los problemas donde se utiliza la búsqueda, siempre está la alternativa de comenzarla desde la meta, buscando el estado inicial. Esto se conoce como *búsqueda por regresión*, y no está claro en que casos representa un mejor algoritmo. Suele argumentarse que hace la búsqueda más guiada al utilizar información sobre la meta. Pero tienen la desventaja de que muchos problemas no tienen las mismas características al verlos en forma regresiva. En la planificación STRIPS hay que tomar en cuenta que la meta no está completamente especificada. Aun así puede definirse la aplicación inversa de una acción, pero los estados que se generan no están todos asociados a planes factibles, por lo que la complejidad de la búsqueda aumenta. HSPr[7] es un planificador regresivo que utiliza h_+. En la figura II.5 podemos ver una definición del espacio de búsqueda de regresión. HSPr busca en dicho espacio con heurística

$$h(e) = \sum_{p \in e} g_{e_0}(p)$$

donde el costo de los átomos $g_{e_0}(p)$ es computado sólo **una** vez desde e_0.

Una heurística admisible en términos de la simplificación que hemos hecho sería asumir que para lograr un conjunto de átomos, hay que lograr el *más* costoso de ellos, y no la suma como asumíamos en h_+. Esta heurística queda definida como sigue:

$$\begin{aligned} g_s^{max}(C) &= \max_{r \in C} g_s(r) \qquad \text{(max costs)} \\ h_{max}(s) &\stackrel{\text{def}}{=} g_s^{max}(M) \end{aligned} \tag{II.5}$$

Pero esta es poco informativa y no ha podido ser utilizada con éxito en búsqueda direccional. Una pregunta interesante es si se pudiera dar una heurística admisible tomando

- los estados e son conjuntos de átomos de A
- el estado inicial tiene los átomos de la meta: $e_0 = M$
- los estados meta $e_m \in E_g$ son tales que $e_m \subseteq I$, puesto que no tienen que llegarse a necesitar todos los atomos del estado inicial. Pero no pueden tenerse otros que no formen parte de I.
- acciones aplicables a en un estado e son las *relevantes* y *consistentes;* es decir, $Adic(a) \cap e \neq \emptyset$ y $Elim(a) \cap e = \emptyset$
- el estado $e' = f(a, e)$ que sigue de la aplicación de una acción aplicable a es tal que $e' = e - Adic(a) + Prec(a)$.
- $c(a, e) = 1$ asociado a cada acción en cada estado

Figura II.5: Modelo de estados del problema de planificación visto como regresión

máximos sobre más de un átomo, de conjuntos de tamaño m. La familia de heurísticas h^m fue introducida en HSPr*[36]. La idea es que h^m aproxime el costo del **conjunto** de átomos C mediante el subconjunto más costoso: $D \subseteq C$, $|D| \leq m$. Esta sería admisible y más informativa que h_{max}, se podría computar polinomialmente para un m fijo, aunque más costosamente mientras m sea más grande.

Partamos de la necesidad de estimar la heurística óptima h^* que de el costo de lograr el conjunto de átomos C desde e_0, que es:

$$h^*(C) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } C \subseteq s_0 \\ \text{mín}_{\langle B,a\rangle \in R(C)}[cost(a) + h^*(B)] & \text{en otro caso} \end{cases}$$

Donde los pares $\langle B, a\rangle \in R(C)$, son aquellos para los cuales se puede llegar a C aplicando la acción a al estado B. Recordando que la aplicación inversa de una acción puede dar transiciones entre estados que no forman parte de ningún plan válido.

Con esta consideración en mente, podemos definir h^m que provee un estimado **admisible** definido por la siguiente relajación de h^*:

$$h^m(C) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } C \subseteq s_0 \\ \text{mín}_{\langle B,a\rangle \in R(C)}[cost(a) + h^*(B)] & \text{si } |C| \leq m \\ \text{máx}_{D \subset C, |D|=m}\ h^m(D) & \text{en otro caso} \end{cases} \quad \text{(II.6)}$$

Esta relajación aproxima el costo de conjuntos grandes C, $|C| > m$, por el costo del subconjunto más costoso $D \subset C$, $|D| \leq m$.

Si usamos $O(p\&q)$ para referirnos al conjunto de acciones que añaden p y q, y $O(p|q)$

el conjunto de acciones que añaden p pero no añaden ni eliminan q. Para $m = 2$, las ecuaciones que definen $h^m(C)$ quedan:

$$h^2(C) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } C \subseteq e_0 \\ \text{mín}_{a \in O(p)}[c(a) + h^2(Prec(a))] & \text{si } C = \{p\} \\ \text{mín} \Big\{ \text{mín}_{a \in O(p\&q)}[c(a) + h^2(Prec(a))] \ , & \\ \quad \text{mín}_{a \in O(p|q)}[c(a) + h^2(Prec(a) \cup \{q\})], & \text{si } C = \{p, q\}; \\ \quad \text{mín}_{a \in O(q|p)} \ [c(a) + h^2(Prec(a) \cup \{p\})] \} & \\ \text{máx}_{\{p,q\} \in C} \, h^2(\{p, q\}) & \text{si } |C| > 2 \end{cases} \tag{II.7}$$

Así, $h^2(C)$ estima el costo de alcanzar C desde el estado inicial e_0. Intuitivamente $h^2(\{p, q\})$ puede ser entendido como el costo mínimo de alcanzar $\{p, q\}$ en una de las siguientes maneras:

- una acción a que añade p y q.
- una acción a añade p pero no añade ni elimina q, pero tenemos que tomar en cuenta la necesidad de establecer q directamente.

Nótese que se toma en cuenta la necesidad de hacer que las acciones sean aplicables, estableciendo antes sus precondiciones.

Para calcular h^2 se utiliza como en casos anteriores un algoritmo similar a *shortest path* donde los nodos son pares de átomos. HSPr* es un planificador óptimo paralelo que utiliza h^2 como heurística, mas regresión e IDA*. En dominios secuenciales tiene buen rendimiento, no así en dominios muy paralelizables. Una discusión sobre esto se encuentra en la sección II.4.

La planificación como búsqueda heurística ha dominado la forma secuencial desde el año 98, en la competencia AIPS. La imposibilidad primera de utilizar búsqueda direccional fue superada gracias a la obtención de heurísticas informativas sintetizadas desde cada problema a resolver. Esta síntesis de heurísticas captura adecuadamente la relación entre las acciones y los átomos, permitiendo evaluar la bondad de los estados. Queda abierto el reto de utilizar heurísticas de este tipo en búsquedas no direccionales, que fue la aproximación dominante hasta el advenimiento de los planificadores actuales y es la forma de ramificación usada por los planificadores herederos de SATPLAN. Kambhampati plantea en REPOP[59] un planificador de orden parcial sub-óptimo que utiliza heurísticas para evaluar los planes y tomar decisiones durante las búsqueda.

II.4. Planificación paralela: planteamiento y soluciones

En un problema de planificación existen acciones que no interfieren entre sí para su aplicación. Se dice que dos acciones están *mutex* (de *MUTual EXclusive*), cuando no pueden ejecutarse en el mismo instante de tiempo, es decir, cuando se cumple la siguiente condición para un par de acciones a y a':

$$Prec(a) \cap (Adic(a') \cup Elim(a')) \neq \emptyset \tag{II.8}$$

En la planificación paralela permitimos que puedan aplicarse al mismo tiempo las acciones que no interfieren entre sí. Un plan en este contexto es una secuencia de conjuntos de acciones: $A_0, A_1, ..., A_{n-1}$. Esto puede ser visto como un modelo[71] Mod que indica cuando es cierto un conjunto de átomos P en cada instante de tiempo i: $Mod \models_i P$. La aplicación de las acciones $A_i = a_1^i, a_2^i, ..., a_m^i$ debe satisfacer las condiciones de la figura II.6.

Las acciones $a_1, \ldots, a_n \in A_i$, deben cumplir:

P1 Para cada par de acciones posible $\langle a, a' \rangle \in A_i$, no están *mutex*

P2 Se cumplen las precondiciones de cada acción:
$Mod \models_i Prec(a_j), j = 0, ..., n$

P3 Se garantizan los efectos positivos en el siguiente instante de tiempo:
$Mod \models_{i+1} Adic(a_j), j = 0, ..., n$

P4 Se garantizan los efectos negativos en el siguiente instante de tiempo:
$Mod \models_{i+1} \neg e_k, \forall e_k \in Elim(a_j), j = 0, ..., n$

P5 Si ninguna acción modifica un átomo, entonces el átomo no cambia entre los dos instantes de tiempos (axioma de cuadro[74]):
$p \notin Adic(a_j) \cup Elim(a_j), j = 0, ..., n \rightarrow (Mod \models_i p \leftrightarrow Mod \models_{i+1} p)$

Figura II.6: Condiciones para acciones en planificación paralela

De manera que un problema de *planificación paralela* es buscar un modelo Mod y conjuntos de acciones $A_0, ..., A_{n-1}$ que satisfacen las condiciones **P1-P5**, satisface el estado inicial en el instante 0 –$Mod \models_0 I$– y satisface la meta en el instante n, $Mod \models_n M$. Existe un modelo Mod si existe un plan paralelo de longitud n.

Con algunas técnicas la planificación paralela permite resolver problemas más complejos, puesto que si tenemos n acciones aplicables en orden indistinto, un planificador direccional consideraría todas las $n!$ combinaciones, mientras que en un plan paralelo es-

tas acciones pueden hacerse todas al mismo tiempo. Por otro lado, cuando se intenta dar una solución paralela con algún tipo de búsqueda direccional, si en planificación secuencial se tenían n acciones por aplicar, en paralela se tiene 2^n opciones, puesto que en cada instante de tiempo se ejecutan un conjunto de acciones. Esto hace especialmente factible el uso de ramificación no direccional, que pueda seleccionar apropiadamente una de esas opciones.

La mayoría de los enfoques actuales para solucionar la planificación paralela tratan de buscar un plan de cierta longitud i, partiendo desde longitud cero, e incrementando en uno cada vez. GRAPHPLAN[4] (ver sección II.4.1) puede ser visto como una búsqueda regresiva sobre una estructura de datos que mantiene información sobre la relación entre acciones y átomos. SATPLAN[41] (ver sección II.4.3) traduce la especificación del problema a un conjunto de formulas proposicionales y, utilizando una herramienta general para satisfacción proposiciona,l busca una asignación factible de las variables que de ser encontrada indica un plan. HSPr* (ver sección II.4.2) usa búsqueda heurística regresiva con una variación de h^2 que es admisible en el dominio paralelo. *Van Beek y Chen*[81], con CPLAN, codifican manualmente varios problemas clásicos de planificación mediante *programación con restricciones* (ver capítulo IV). Tienen resultados interesantes pero no son independientes del dominio, ya que cada caso requiere intervención directa. A continuación veremos más en detalles varios de estos planificadores.

II.4.1. GRAPHPLAN

Con GRAPHPLAN[4] se da un salto importante en el rendimiento de los planificadores. Este genera un grafo que contiene información sobre las acciones aplicables, los conflictos entre ellos; y allí hace búsqueda para generar los planes.

Un *grafo de plan* tiene capas proposicionales y de acciones: P_i, A_i, $i = 0, ..., n$.

- P_0 contiene los átomos ciertos en I
- A_i contiene las acciones cuyas precondiciones son ciertas en P_i
- P_{i+1} contiene los efectos positivos de las acciones en A_i
- Los mutexes (Eq. II.8) son extendidos para referirse a pares de acciones y proposiciones:
  - p y q en P_i están mutex si todas las acciones en A_i que soportan p y q están mutex en A_{i-1}

- a y a' en A_i están mutex si están mutex según Eq. II.8 o $Prec(a) \cup Prec(a')$ contiene un mutex en P_i

El algoritmo empieza expandiendo el grafo hasta que aparezcan todos los átomos de M sin ningún mutex entre ellos. Luego se intenta una búsqueda regresiva hasta obtener el estado inicial. Si dicha búsqueda falla, se expande el grafo un nivel más y se repite el proceso. La búsqueda para una meta B se da como sigue, partiendo desde la profundidad n:

1. si $n = 0$. Si la meta incluye el estado inicial ($B \subseteq I$), triunfar; sino, fallar.
2. sino, buscar con meta = $Precs(A)$ y profundidad $n-1$, donde A es un conjunto de acciones compatibles entre sí en la capa A_{n-1}, elegido de tal manera que todos los átomos de la meta B sean agregados por O.

Un problema al buscar siempre una acción para que agregue un átomo, es que algunos átomos permanecen porque fueron agregados en cierto momento. Es por ello que GRAPHPLAN usa un conjunto de acciones llamadas *no-op*. Para cada átomo p, $a = no - op(p)$ cumple: $Prec(a) = Add(a) = \{p\}, Del(a) = \emptyset$. También sucede que se recorre varias veces el mismo espacio, al ser la búsqueda IDA*, por lo que se utiliza *memoización* para recordar zonas del espacio que ya fueron exploradas.

Interesantemente, si no extendemos los mutex el nivel i de la capa donde aparece por primera vez los átomos corresponde al valor de la heurística h^1 (Eq. II.5). Extendiendo los mutex, el nivel de un par de átomos coincide con una variación de h^2 (Eq. II.7). Esta variación permite que sea admisible en planificación paralela, siendo la usada por HSPr* (ver sección II.4.2).

Con estas consideraciones podríamos mirar GRAPHPLAN como una búsqueda heurística: IDA* con una heurística similar a h^2[36]. El grafo del plan permite además que nunca se consideren opciones que conllevan a falla.

Sobre GRAPHPLAN se han hecho diferentes versiones que mejoran la síntesis de grafo y variaciones en la búsqueda[49]. Incluso versiones para manejo de lenguajes más expresivos como ADL(IPP [43]), o incluir acciones de observación del ambiente[84]

II.4.2. HSPr*

El planificador HSPr*[36] incluye la posibilidad de retornar planes óptimos paralelos. Para ello se modifica la definición de h^2 (Eq. II.7) a fin de que sea admisible en dicho contexto. La reformulación paralela h^2_p es:

$$h_p^2(C) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } C \subseteq e_0 \\ \text{mín}_{a \in O(p)}[c(a) + h_p^2(Prec(a))] & \text{si } C = \{p\} \\ \text{mín} \Big\{ \text{mín}_{a \in O(p\&q)}[c(a) + h_p^2(Prec(a))] \, , & \\ \quad \text{mín}_{a \in O(p|q)}[c(a) + h_p^2(Prec(a) \cup \{q\})], & \text{si } C = \{p, q\} \\ \quad \text{mín}_{a \in O(q|p)}[c(a) + h_p^2(Prec(a) \cup \{p\})] & \\ \quad \text{mín}_{\langle a, a' \rangle \in O(p,q)} \, [c(a) + h_p^2(Prec(a) \cup Prec(a'))] \Big\} & \\ \text{máx}_{\{p,q\} \in C} \, h_p^2(\{p, q\}) & \text{si } |C| > 2 \end{cases} \quad \text{(II.9)}$$

$O(p, q)$ son los pares de acciones que añaden p y q. La única modificación es que tambien se minimiza sobre las acciones paralelas para lograr un par de átomos: $\text{mín}_{\langle a,a' \rangle \in O(p,q)}[c(a) + h_p^2(Prec(a) \cup Prec(a'))]$. Esto, suponiendo el costo de a igual al costo de a'. El argumento informal relativo a esta parte de la ecuación, que completaría la descripción que se dio al final de la sección II.3.2, página 30, sería:

- un par de acciones $a\&a'$ que pueden ser aplicados en *paralelo* y establecer el par de átomos $p\&q$ al costo de una acción primitiva

h^2 (Eq. II.7) es equivalente a la heurística usada por GraphPlan, asumiendo que el costo de todas las acciones es igual, y que todas las acciones están *mutex*, es decir, que no hay paralelismo entre ellas. En cambio, se puede demostrar que h_p^2 coincide *exactamente* con el nivel en que aparece el par de átomos en el grafo del plan[36]. h_p^2 se calcula sin necesidad del grafo del plan, con lo que no tiene los problemas de memoria que muchas veces limitan la usabilidad de GraphPlan. Pero el grafo tiene otras ventajas, puesto que permite buscar sólo dentro del espacio factible y además mantiene la relación entre acciones y átomos en cada capa, lo que puede usarse para otras optimizaciones.

hspr* paralelo puede resolver problemas que la versión serial no, aunque es más lento que GraphPlan, en particular en dominios altamente paralelos. Esto se debe principalmente a que GraphPlan usa el grafo para seleccionar las acciones, mientras que la manera en que lo hace hspr* mantiene muchas posibilidades equivalentes entre sí, que se consideran individualmente.

II.4.3. SATPlan

SATPlan[41] lleva el problema a un conjunto de fórmulas lógicas al que se le da solución con motores de inferencia proposicionales. Para ello, lleva un problema de planificación $\langle P, I, M, A \rangle$ con una horizonte n a una teoría proposicional donde se supone que la

meta se va a lograr en el instante n. Esta teoría incluye variables $p_0, p_1, ..., p_n + 1$ que son ciertas si el átomo p es cierto en los instantes $0, 1, ..., n + 1$. Similarmente, incluye variable $o_0, o_1, ..., o_n$ que son ciertas si la acción a se aplica en los instantes $0, 1, ..., n$. Si existe una asignación factible para dichas variables en la teoría, entonces los variables ciertas asociadas a las acciones indican un plan que lleva a la meta en n pasos. Sino se consigue asignación con horizonte n, entonces se intenta con $n + 1$.

La teoría para un problema $\langle P, I, M, A \rangle$ es:

1. Inicio: p_0 para $p \in I$, $\neg q_0$ para $q \notin I$
2. Meta: p_{n+1} para $p \in M$
3. Acciones: $i = 0, 1, ..., n - 1$

   $a_i \supset p_i$ para $p \in Prec(a)$

   $a_i \supset p_{i+1}$ para $p \in Adic(a)$

   $a_i \supset \neg p_{i+1}$ para $p \in Elim(a)$
4. Cuadro: $(\bigwedge_{a:p \in Adic(a) \cup Elim(a)}) \supset p_i \equiv p_{i+1}$
5. Mutex: si a y a' están mutex: $\neg(a_i \wedge a'_i)$

Cuadro se refiere a un axioma de cuadro[72], el cual se utiliza para conservar los atomos que no están siendo tocados por ninguna acción. Un problema clásico dentro del estudio de los lenguajes de representación para acciones, era el problema de no tener que sobreabundar en axiomas donde se especificara que de no haber cambios en una proposición, esta debía conservarse entre instantes de tiempo (*Frame Problem*[74]). Luego de un tiempo se llego a una manera de resolver la interrogante, y la forma de la clausula de cuadro que se agrega en la teoría refleja tales resultados.

Frente a esta teoría constituida por un conjunto de clausulas, se utiliza una herramienta para solucionar problemas de satisfactibilidad proposicional. Kautz y Selman incluso plantearon que la planificación en si misma iría perdiendo espacio como problema en sí mismo, puesto que la satisfacción proposicional era *la* manera de atacar el problema. Las herramientas exactas que retornan una asignación de existir, o reportan la imposibilidad de encontrarla, se basan en el clásico algoritmo de Davis-Putnam[21]. Este funciona seleccionando una variable bajo ciertos criterios y luego intentando la opción de que sea cierta o falsa, continuando las búsqueda recursivamente. Estas decisiones en cada rama fuerzan a que un átomo o una acción sean ciertos en algún instante de tiempo. El uso de SAT esta alejado de la búsqueda direccional, y guarda relación con la manera de tomar decisiones en cualquier punto del plan que se usa en planificación de orden parcial (sección II.2.1).

Este esquema permite también hacer planificación sub-óptima, utilizando herramientas incompletas que tratan de encontrar aleatoriamente asignaciones para las variables. Luego de un número de iteraciones de este proceso sin haber encontrado una asignación, se puede reportar que no es satisfactible, e intentar con un horizonte mayor.

En BlackBox[42] se usa la idea de SATPlan pero codificando también el grafo del plan de GraphPlan. De esta manera la teoría es más informativa. Esto, y el uso de las mejores herramientas disponibles para la satisfacción de restricciones, hacen de BlackBox el mejor planificador en dominio paralelo reportado hasta ahora.

Una posible desventaja de esta aproximación es que se están perdiendo información al reducir de un lenguaje más expresivo (STRIPS) a uno más simple. De hecho, HSP hace factible la planificación como búsqueda porque lograr capturar en forma de una heurística, parte de la estructura del problema a resolver.

Rintanen[70] estudia las inferencias que hacen las herramientas sobre las teorías que se usan en SATPlan y reporta un planificador que usa estas inferencias directamente, sin tener que disponer de una herramienta general. Obtiene resultados comparables, aunque peores que SATPlan, con mucho menor uso de memoria y con mayores posibilidades de usar más conocimiento del problema específico de planificación.

II.5. Planificación Temporal

Planificación temporal es extender la planificación clásica añadiendo duración de las acciones y permitiendo que se ejecuten de manera concurrente[32]. A continuación presentaremos sólo la definición del problema, puesto que no será la tarea que resolveremos, pero nuestra formulación contiene ideas que podrían usarse en este dominio.

En primer lugar, consideremos que para toda acción a tenemos una duración $d(a)$ y tenemos un predicado $comp(a, a')$ que indica cuando las acciones a y a' pueden ejecutarse concurrentemente. Así, *comp* puede se usado para representar condiciones de mutua exclusión (mutex, Eq. II.8). En investigación de operaciones es común que las tareas por hacer necesiten ciertos recursos. Cuando se tiene el recurso necesario para que que se haga sólo una tarea, hablamos de *recursos unitarios*, en ese caso si un conjunto de acciones A necesitan dicho recursos, podemos indicar en *comp* que dichas tareas no pueden ejecutarse concurrentemente.

Para obtener una aproximación a un modelo de estados para planificación temporal podemos basarnos en el introducido para la planificación paralela (ver [32]). Tenemos de igual modo una secuencia de conjuntos de acciones: $A_0, A_1, ..., A_{n-1}$. Y un modelo Mod que indica cuando es cierto un conjunto de átomos P en cada instante de tiempo i: $Mod \models_i P$.

La aplicación de las acciones $A_i = a_i^1, a_i^2, ..., a_i^m$ cumple con las mismas condiciones de la figura II.6. Pero el tiempo en este contexto no es una sucesión consecutiva de instantes de tiempo. En cambio, tenemos una secuencia $t_0, t_1, ..., t_n$ de tiempos, tal que t_i es el tiempo en el cual todas las A_i comienza su ejecución. El momento de culminación de una acción a en A_i es $t_i + d(a)$. Los tiempos de inicio son $t_0 = 0$, y t_{i+1} es el momento de culminación de la primera acción en $A_0, A_1, ..., A_i$ que termina después de t_i. Nótese que este modelo esta evitando que puedan generarse planes que tengan espacios de tiempos sin ejecutar ninguna acción, pero esta ha resultado no ser una restricción fuerte. De nuevo, tiene sentido minimizar el tiempo de completación del plan temporal. Esto es, el mayor momento de culminación de las acciones en A_{n-1}.

En este dominio, como en el paralelo, tampoco ha funcionado muy bien la planificación como búsqueda heurística. A pesar de ello Haslum y Geffner[37] reportan un planificador temporal con recursos que usa búsqueda heurística. Un problema sigue siendo el número de opciones en la ramificación, conocido en inglés como *branching factor*. En dicho trabajo puede verse en detalle un modelo de estados para el problema.

Posiblemente enfoques como SATPLAN u otros basados en restricciones puedan ser usados con heurísticas para podar adecuadamente el espacio de búsqueda y dar mejores rendimientos. Este trabajo es un paso en esa dirección. Existen planificadores temporales basados en POP, como RAX[40], aunque para dar rendimientos aceptables requieren mucha entonación y conocimiento del dominio.

II.6. Situación actual

De todos estos avances hay varias ideas que conviene destacar del conjunto. Una es la importancia de *calcular cotas inferiores* para el costo de lograr ciertos objetivos. Esto permite podar drásticamente el espacio de búsqueda. Esta idea no es nueva, pero si lo es el poder *sintetizarla* a partir de la definición del problema. Esto ha sido usado principalmente en planificadores que buscan en el espacio de estados. Cuando los problemas son inherentemente secuenciales, los algoritmos más exitosos en búsqueda[46] son los que mejor escalan, pero tienen problemas para tratar los casos de planificación paralela[36].

En los dominios con paralelismo tiene menos sentido ordenar las acciones a realizar, con lo cual el espacio de búsqueda sobre estados podría crecer artificialmente. Además, podría tener ventajas tomar la decisión de colocar una acción en cualquier lugar del plan, tal como se hace en los planificadores basados en satisfacción de restricciones o de orden parcial. En este caso se busca en el espacio de planes. Sin embargo, con algunas excepciones (REPOP[59] y UNPOP[53]), este tipo de planificadores no tienen cotas inferiores que sirvan

para reducir el espacio de búsqueda.

Hemos visto diferentes maneras de recorrer el espacio. En el caso direccional, se trata de completar el principio o final del plan, partiendo de que se tiene la otra parte (ver sección II.3). En casos como SATPlan y otros basados en restricciones, se tiene variables para las acciones en el tiempo (ver sección II.4.3. Una manera de recorrer ese espacio es obligar a que una acción este o no en un instante de tiempo. En POP (ver sección II.2.1) se colocan restricciones de precedencias para resolver conflictos e ir reparando el plan.

Así, está abierta la pregunta sobre como atacar la planificación paralela óptima, y también la temporal con duraciones y recursos unitarios, usando la búsqueda heurística que tan buenos resultados ha dado en otros contextos. La respuesta podría ser la búsqueda heurística en el espacio de planes. Por ahora, los mejores planificadores paralelos óptimos pertenecen a las familias GraphPlan y SATPlan.

Capítulo III

RAMIFICACIÓN Y PODA EN OPTIMIZACIÓN COMBINATORIA

El problema de planificación está emparentado con varios problemas estudiados en Investigación de Operaciones. En este capítulo estudiaremos dos problemas cuyas soluciones mediante ramificación y poda contiene lecciones útiles para resolver el problema que nos atañe.

La noción de *cotas inferiores* usadas en Optimización Combinatoria[2, 18] es familiar en la **IA** donde es llamada *heurística admisible*[58, 61]. Pero la noción de ramificación es mucho menos familiar. Esto está probablemente influido porque en la resolución de muchos problemas, las ramas examinadas corresponden a la *aplicación de acciones*. En problemas como el 8-puzzle, 15-puzzle o el cubo de *Rubik*, un estado es *expandido* en la búsqueda aplicando todas las posibles acciones, como en la planificación mediante búsqueda heurística (ver sección II.3). Esta manera de explorar diversas ramas de la búsqueda la llamaremos *ramificación direccional* o *ramificación en las acciones*. A pesar de ser tan frecuentemente usada no es siempre la mejor manera. A continuación veremos algunos problemas de optimización combinatoria que usan esquemas de ramificación no direccional, usando también el cálculo de cotas inferiores.

III.1. El Problema del Agente Viajero mediante Ramificación y Poda

Este problema, conocido en ingles como *Traveling Salesman Problem* (TSP), consiste en encontrar una ruta con costo mínimo que recorra un conjunto de ciudades[2], y ha sido ampliamente estudiado. Se trabaja en algoritmos para solucionar a nivel práctico problemas de cada vez mayor complejidad, puesto que es *NP-completo*. Una variante muy estudiada es el TSP asimétrico (ATSP), donde el costo de viajar entre un par de ciudades no es necesariamente simétrico. Una manera bastante exitosa de aproximarse a él involucra

1. Los estados σ son ATSPs
2. El estado inicial σ_0 es el ATSP original
3. La cota inferior $f(\sigma)$ es calculada solucionando la relajación al problema de asignación de σ; obteniendo se una o mas subrutas
4. Los estados meta son aquellos cuya relajación equivale a una sola ruta
5. Los estados no terminales σ son expandidos seleccionando un arco $i \to j$ de una subruta y generando dos hijos:
   - σ_{ij}^{+}: fuerza que el arco $i \to j$ esté en la solución. Se obtiene excluyendo los enlaces $i \to j'$ y $i' \to j$ para todo $j' \neq j$ y $i' \neq i$, siendo que sólo un arco sale de i, y un otro único llega a j, respectivamente
   - σ_{ij}^{-}: excluye el arco $i \to j$ de la solución. Se obtiene excluyendo el arco $i \to j$ de σ

Figura III.1: Esquema de Ramificación y Poda para ATSPs

solucionar una versión simplificada del problema, conocido como *el problema de asignación* [2, 87]. Éste consiste en asignar cada cuidad i a una única ciudad $siguiente(i) = j$ tal que la suma de las distancias c_{ij} desde la ciudad i a la j es minimizada. Tal asignación puede resultar en una única ruta que recorra todas las ciudades, o varias rutas disjuntas. En cualquier caso, el costo de la asignación es una *cota inferior* al costo de un ATSP. En el primero caso, además, esta cota inferior es *exactamente* una asignación óptima que representa una ruta que recorre en la menor distancia todas las ciudades. Estas propiedades pueden usarse en el contexto de un algoritmo de *Ramificación y Poda* para buscar soluciones óptimas al ATSP. Diremos que un arco $i \to j$ corresponde a una ruta directa entre la ciudad i y la j.

En la ramificación y poda se tiene una manera de obtener nodos hijos a partir de cada nodo, lo que permite recorrer el espacio de estado hasta encontrar el nodo con costo óptimo. En la figura III.1 puede verse un esquema de ramificación y poda para ATSP. [1]

III.2. Algoritmos para Ramificación y Poda

Este esquema de ramificación y poda puede ser implementado mediante diferentes algoritmos tales como IDA* o *Búsqueda en Profundidad iterativa*[45]. Por ejemplo, un IDA*

[1]Este esquema puede mejorarse, notando que la relajación genera la misma asignación y, por lo tanto, las mismas rutas disjuntas, tanto para σ como para σ^{+}, puesto que el arco ya esta incluido (ver [2, 87]).

completo para el TSP puede ser obtenido definiendo la selección del arco $i \rightarrow j$ en el cual se va a ramificar, y el orden en el cual las dos ramas son consideradas (Paso 5 en la figura III.1). En todos los algoritmos de ramificación y poda, la acción principal es evaluar la *condición de poda* $f(\sigma) \leq C$ para un costo límite C, y difieren entre ellos en el estado que seleccionan para expandir (*e.g.* buscar en profundidad, el menos costoso primero, etc.), o en la manera en que el límite es ajustado (decrementado hasta que no hay más soluciones, incrementado hasta que se encuentra una solución). Similares consideraciones aplican para los algoritmos generales de búsqueda[72].

Para encontrar una solución óptima, un esquema de ramificación y poda debe ser *correcto* y *completo* en este sentido: los estados meta deben representar soluciones, y algunos estados deben representar soluciones *óptimas*. Además de permitir, por supuesto, recorrer todo el espacio de búsqueda. Esto permite llegar sistemáticamente a todas las soluciones y, por la manera en que se recorre el espacio, poder reportar una que sea óptima. Con estas condiciones, cualquier algoritmo usable para ramificación y poda garantizará el encontrar una solución óptima.

El que los estados metas representen soluciones es fácil de verificar en el esquema de la figura III.1: todos los estados terminales representan rutas completas y por lo tanto son una solución del problema. Las ramificaciones no pueden excluir soluciones óptimas, ya que una solución óptima para σ será una solución óptima para todos hijos; y si σ es un estado terminal, entonces la relajación al problema de asignación da una óptima solución. Luego, el esquema de ramificación y poda es correcto y completo en el sentido anterior.

Note que la ramificación usada no es aplicar las acciones de ir de una ciudad a uno de sus vecinos, sino haciendo *compromisos*. Es decir, que se asume que ciertos arcos están dentro o fuera de la solución. Estos compromisos forman una ruta parcial de manera muy parecida a los enlaces causales y restricciones de precedencia usados en los planes parciales (ver sección II.2.1). Ambas son formas de ramificación no direccional o *ramificación con compromisos*, en contraste con la *ramificación en acciones*. Por supuesto, ambas son compatibles con el uso de cotas inferiores para podar el espacio de búsqueda.

III.3. Job Shop Scheduling

El Job Shop es relevante en nuestra discusión por dos razones. En primer lugar, como el ATSP, provee un dominio claro y bien entendido donde es factible hacer ramificación en acciones aunque no sea la manera más efectiva de atacar el problema. Segundo, el Job Shop esta fuertemente relacionado con el problema de planificación temporal, por lo que las ideas con utilidad aquí son relevantes.

El problema del Job Shop (JSP) consta de un conjunto de trabajos $j_1, \ldots, j_m$, donde cada uno consta de una cadena de tareas $t_{i1}, \ldots, t_{in}$, $i = 1, \ldots, m$, con una duración $D(t_{ij})$ que deben ser ejecutadas en orden sobre un conjunto de n recursos unarios $R(t_{ij})$[63]. Una *planificación factible* es una asignación de tiempos $T(t_{ij})$ a cada tarea t_{ij} tal que las restricciones de precedencia en las tareas y las restricciones sobre el uso de recurso se satisfacen. Tales restricciones pueden ser escritas como

$$\begin{array}{l}\text{Para } i \in [1 \ldots m], j \in [1 \ldots n-1]: \\ \qquad t_{ij} \prec t_{i,j+1} \qquad \text{(restricciones de precedencia)}\end{array} \tag{III.1}$$

$$\begin{array}{l}\text{Para } i,k \in [1 \ldots m], j,l \in [1 \ldots n]: \\ R(t_{ij}) = R(t_{kl}) \;\Rightarrow\; t_{ij} \prec t_{kl} \;\vee\; t_{kl} \prec t_{ij} \quad \text{(restricciones de recursos)}\end{array} \tag{III.2}$$

donde

$$t \prec t' \text{ significa } T(t) + D(t) \leq T(t')$$

Una *planificación óptima* es una planificación factible con el *menor tiempo de completación*, esto es, el instante de tiempo en el cual todas las tareas han sido completadas.

También para este problema puede formularse un esquema de ramificación en el cual las acciones (conjuntos de tareas realizadas paralelamente) son aplicadas hacia adelante o hacia atrás. Un problema con esto es que hay demasiadas opciones entre las cuales decidir. La cantidad de opciones que pueden tomarse en un esquema de ramificación se conoce como *factor de ramificación* (en ingles, *branching factor*). Si este es demasiado alto, el rendimiento de los algoritmos de búsqueda suele empeorar. Siguiendo la manera de aproximarse de *Cesta et al*[14], que extiende ideas de *Calier et al*[10] y *Cheng et al*[79], formularemos un esquema no direccional. En este, cada estado σ es un JSP relajado que incluye todas las restricciones de precedencia en el JSP original, pero *sin* restricciones de recursos. El problema relajado corresponde a un *Problema Temporal Simple* (*Simple Temporal Problem*, STP)[22], el cual es tratable y puede resolverse mediante una variedad de algoritmos de ruta más corta (*shortest path*, [22, 13, 33]) o de propagación de restricciones (ver capítulo IV). El algoritmo determina la consistencia de un STP σ y si lo es, retorna una cota inferior $h_\sigma(t)$ del momento en el cual la tarea t puede empezar, y por lo tanto, tenemos inmediatamete una cota inferior $f(\sigma)$ que es el tiempo de completación de todas las tareas. Nos referiremos a la planificación en la cual cada tarea t_{ij} es ejecutada en el momento indicado por su cota inferior $h_\sigma(t_{ij})$ como la *planificación relajada* y la denotaremos h_σ. Es claro que la planificación relajada satisface el STP σ, y por lo tanto, es una solución al JSP original si y solo si satisface también las restricciones de recursos

1. Los estados σ son problemas temporales simples (STPs)
2. El estado inicial σ_0 dado por las restricciones de precedencias en las tareas
3. Las cotas inferiores $f(\sigma)$ y las planificaciones relajadas h_σ son obtenidas resolviendo el STP σ; donde $f(\sigma) = \infty$ si el STP es inconsistente
4. Los estados terminales son de dos tipos:
   - σ es un *punto muerto* si $f(\sigma) = \infty$
   - Un estado meta si ningún par de tareas están en conflicto en h_σ
5. Los hijos son generados a partir de σ seleccionando una par de tareas en conflicto t y t', y generando dos hijos:
   - $\sigma + \{t \prec t'\}$
   - $\sigma + \{t' \prec t\}$

Figura III.2: Esquema de Ramificación y Poda para el Job Shop

(Eq. III.2). En otro caso, debe haber un par de tareas t y t' que estén *conflicto* en σ, esto es, tareas que usan el mismo recurso y cuya ejecución se solapa en h_σ. Esto sugiere un esquema de ramificación y poda mostrada en la figura III.2.

Es simple verificar que este esquema también es correcto y completo en el sentido definido anteriormente: lo estados meta son soluciones al problema, y alguno de ellos son soluciones óptimas.

Como en el esquema para el ATSP, este tiene muchas de las características que forman un poderoso esquema de ramificación y poda: fuerte integración entre ramificación y cálculo de cotas inferiores, pocos hijos, un rápido cálculo de cotas inferiores, que además es susceptible a ser calculado incrementalmente para mayor rapidez durante la búsqueda.

Capítulo IV

PROGRAMACIÓN CON RESTRICCIONES

Partamos del Job Shop, estudiado en el capítulo anterior. En la tabla IV.1 tenemos información sobre algunas tareas por realizar (tomado de [51], p. 273).

Allí se especifica un conjunto de tareas, cada una con una duración. Algunas de ellas requieren que antes se realice otra, o utilizan algún recurso unitario. Definamos variables T_i que indican el momento de inicio para cada tarea t_i. La solución a este problema es una asignación factible a dichas variables. Podemos plantear varias restricciones sobre estos tiempos de inicio, basados en la definición del problema. Supondremos que tenemos una tarea *Final* que usamos para indicar el final de todas ellas.

- Cada tarea comienza luego del primer instante, y comienza antes del final de acuerdo a su duración:
  $0 \leq T_i \wedge T_i + duracion(i) \leq Final$, para cada tarea
- La tarea i tiene que comenzar antes que la tarea j, si $i \prec j$:
  $T_i + duracion(i) \leq T_j$
- Si i y j usan el mismo recurso maq, no pueden ejecutarse al mismo tiempo:
  $T_i \geq T_j + duracion(j) \vee T_j \geq T_i + duracion(i)$

Tarea	Duración	precedida por	recurso que usa
t_1	3		maq_1
t_2	8		maq_1
t_3	8	t_4, t_5	maq_1
t_4	6		maq_2
t_5	3	t_1	maq_2
t_6	4	t_1	maq_2

Cuadro IV.1: Datos de un problema de *Scheduling*

Estas restricciones caracterizan una solución *factible*. Sería interesante tener un mecanismo que permita generar asignaciones factibles, para usarse en el contexto de buscar una solución *óptima* al problema.

Un *problema de satisfacción de restricciones* (PSR) es un tripleta $\langle V, D, R\rangle$, respectivamente las variables, los dominios, y las restricciones. Cada variable tiene un dominio finito asociado. Por ejemplo, el problema anterior quedaría:

$$\begin{aligned}\langle V, D, R\rangle \quad = \quad & \langle \{t_1, t_2, ...\}, \{[0.,20], [0.,30], ...\}, \\ & \{0 \leq T_6 \wedge T_6 + 4 \leq end, T_6 \geq T_1 + 3, T_6 \geq T_4 + 6 \vee T_4 \geq T_6 + 4, ...\}\rangle\end{aligned}$$

Una *solución* es una asignación de valores a variables del dominio que satisfacen todas las restricciones. Un PSR es *consistente* si existe al menos una solución para este. Note que un dominio vacío para alguna variable implica que el problema es inconsistente, puesto que para dicha variable no hay valor posible que forme parte de una solución al problema.

En la *programación con restricciones*[51] se especifican variables y restricciones para modelar situaciones por medio de una librería o lenguaje de programación. Este se resuelve usando algoritmos generales para PSR o algoritmos específicos al tipo de restricción que se esté usando, tomando decisiones concretas en base al problema que se pretende resolver. En los PSR's[72] estudiados desde hace años dentro de la IA, se plantea solucionar estos problemas de manera independiente a las restricciones especificas que se estén manejando. Mientras que en la programación con restricciones se opta por contar con la semántica específica para las restricciones, y se optimiza el rendimiento en base a ellas. Los métodos utilizados en satisfacción de restricciones son relevantes en *programación*, pero además las optimizaciones dependientes de la semántica que ellos puedan dar afectan drásticamente el rendimiento de la búsqueda de soluciones a los problemas que se modelen. Algunos problemas que se formulan y resuelven con programación con restricciones incluyen:

- N-reinas
- Coloración de grafos
- Matrimonio estable
- Job-Shop
- Asignación de recursos

Las variables en programación con restricciones pueden tener diferentes dominios: reales, racionales, intervalos, booleanos, árboles. Aquí nos ocuparemos de los dominios enteros finitos. En ese contexto, dentro de las restricciones se pueden usar símbolos aritméticos, desigualdades. En particular, las restricciones que se usaron en el ejemplo con que comenzó este capítulo son válidas dentro de la mayoría de los lenguajes o librerías existentes. Podrían también tenerse restricciones definidas por extensión que especifican cada valor del dominio que cumple con la restricción.

IV.1. Algoritmos

La pregunta inmediata es cómo calcular una o más soluciones factibles. Una posibilidad inmediata, conocida como *vuelta atrás cronológica* (*chronological backtraking*), es para cada variable, ir instanciandola en cada valor posible. Eso equivale a hacer una *búsqueda en profundidad* donde la acción que lleva de un nodo al otro es la instanciación de una variable. Obsérvese que dicha búsqueda es completa -siempre retorna una solución- puesto que el número de variables y valores es finito. El problema es que, en principio, dicha búsqueda es completamente ciega.

Otra aproximación es resolver de forma incompleta y rápida el problema. Podrían definirse formas de inferencias limitadas que detecten rápidamente ciertas inconsistencias, y puedan reducir el dominio de ciertas variables a fin de limitar el espacio donde hacer la búsqueda. Esta técnica se conoce como *propagación de restricciones*. Estás técnicas son incompletas y puede retornar una solución, reportar infactibilidad o una reducción del problema. Pero su cálculo requiere tiempo polinomial en contraste con la solución completa que requiere tiempo exponencial.

IV.1.1. Consistencia de nodo y de arco

Llamemos $D(x)$ al dominio de la variable x. Diremos que una restricción r es *consistente de nodo* si tiene más de una variable $|vars(r)| > 1$ o, siendo x la única variable, cada valor posible satisface la restricción: $\forall X \in D(x)\,(x = X \rightarrow c)$. Un conjunto de restricciones R es consistente de nodo si cada restricción r_i es consistente de nodo.

Una restricción r es *consistente de arco* si el número de variables que tiene es diferente de 2 $|vars(r)| \neq 2$, o si $vars(r) = x, y$, entonces para cada $d_x \in D(x)$, existe algún $d_y \in D(y)$ tal que $\{x \mapsto d_x, y \mapsto d_y\}$ es una solución para r, y para cada $d_y \in D(y)$, existe algún $d_x \in D(x)$ tal que $\{x \mapsto d_x, y \mapsto d_y\}$ es una solución para r. Un conjunto de restricciones R es consistente de arco si cada restricción r_i es consistente de arco.

Considere un PSR con variables A, B, C, D, cada una de ellas con dominio $\{1, 2, 3\}$, y las siguientes restricciones sobre dichas variables:

$$A \neq B \wedge A \neq C \wedge B \neq D \wedge C \neq D$$

Este PSR es consistente de nodo, puesto que no hay ninguna restricción unaria. Las binarias también son consistentes de arco. Por ejemplo para la variable A tenemos las restricciones binarias $A \neq B \wedge A \neq C$. Para cada valor posible para A, existe un valor posible en la otra variable. Por ejemplo, si $A = 1$, un valor posible para B y C es 2. Pero si $D(A) = \{1\}$ entonces estas dos restricciones no serían consistentes de nodo, aunque podríamos lograrlo fácilmente haciendo $D(B) = D(C) = \{2, 3\}$.

Esto sugiere la idea de que podemos utilizar las nociones de consistencia para construir algoritmos que partiendo de un PSR se obtenga uno equivalente que sea consistente de nodo o de arco. Nótese que esto no elimina soluciones, siendo que los valores eliminados no forman parte de ninguna solución factible. Considerando todos los dominios como $\{1, 2\}$, el PSR no es consistente de arco, aunque si es consistente de nodo. Otro caso interesante en este ejemplo es asumir que el dominio de todas las variables es $\{1\}$, en el que no hay ninguna asignación posible. En este caso, si optamos por eliminar los valores que no pueden ser parte de una asignación por violar la consistencia de nodo, obtendremos un dominio vacío para alguna variable. Un PSR con alguna variable con dominio vacío es *inconsistente*.

Obtener consistencia de nodo es trivial, recorriendo cada restricción unitaria, y eliminando los valores que no correspondan. Pero en el caso de la consistencia de arco, el eliminar un valor de un dominio puede provocar que los dominios de otras variables resulten afectados. Para ello se toman las restricciones, y se verifica consistencia de arco de cada una, eliminando los valores que no la cumplan. Se siguen recorriendo todas las restricciones hasta que se alcanza punto fijo. En la figura IV.1 puede verse un algoritmo que logra la consistencia de arco de un problema. Un problema con esta aproximación es que se revisan muchas restricciones irrelevantes. Una manera es mantener las restricciones que necesitan ser revisadas. El algoritmo **AC-3**[51] usa esta idea. Sigue siendo uno de los más usados puesto que, además de las restricciones, no mantiene ninguna estructura de datos para acelerar el cálculo. Tampoco supone ninguna forma particular de las restricciones.

Si *consistencia-de-arco* de un PSR retorna una dominio vacío para alguna variable, el PSR es inconsistente; si retorna una valuación para cada variable, esa es una solución; y sino, podría ser inconsistente o no. Siendo que este algoritmo retorna nuevos dominios para las variables, puede usarse en combinación con otros algoritmos de consistencia, como *consistencia-de-nodo*, para obtener mayores reducciones en los dominios.

arco-primitiva(r: Restriccion, D: Dominios)
si $|variables(r)| = 2$ entonces
  sea $\{x, y\} = variables(r)$
  eliminar $d_x \in D(x)$ si para ningun $d_y \in D(y), \{x \mapsto d_x, y \mapsto d_y\}$ es una solucion para r
  eliminar $d_y \in D(y)$ si para ningun $d_x \in D(x), \{x \mapsto d_x, y \mapsto d_y\}$ es una solucion para r
retornar D

consistencia-de-arco(R: Restricciones, D: Dominios)
Repetir
  W $\leftarrow$ D
  Para cada $r_i \in R$
    D $\leftarrow$ arco-primitiva(r_i, D)
Hasta D $\equiv$ W
retornar D

Figura IV.1: Algoritmo: *consistencia de arco* en un PSR

IV.1.2. Otras formas de consistencia

Una limitación de las anteriores formas de consistencia es que trabajan sólo sobre restricciones unarias o binarias. Pueden definirse otras nociones generales de consistencia para relaciones de aridad mayor que 2, pero determinar la *hiper-consistencia* de un PSR es *NP-hard*[51], tanto como obtener satisfactibilidad del PSR en sí mismo.

Un PSR es *aritmético* cuando los dominios de las variables son enteros, y las restricciones primitivas son aritméticas. La gran mayoría de los PSR usados en la literatura y en la industria pertenecen a esta clase. Muchos problemas que no involucran inicialmente dominios enteros, pueden modelarse como aritméticos. Pueden usarse los métodos de consistencia de arco y nodo con estos, pero conociendo la naturaleza de las variables y de las restricciones podemos definir un nuevo tipo de consistencia: *consistencia de frontera*. Para ello limitamos los dominios a su *mínimo* y *máximo*. Un *rango* $[l..u]$ representa el conjunto de enteros $\{l, l+1, \ldots, u\}$ si $l \leq u$, en otro caso representa un conjunto vacío. Llamaremos $min_D(x)$ y $max_D(x)$ al elemento mínimo y máximo, respectivamente, del dominio de x, $D(X)$.

Así, diremos que una restricción primitiva r es *consistente de frontera*([51], p. 98) con dominio D si para cada variable $x \in vars(r)$, existe:

- una asignación de números reales, $d_1, d_2, \ldots, d_k$, para el resto de las variables de la restricción, $x_1, x_2, \ldots, x_k$, tal que $min_D(x_j) \leq d_j \leq max_D(x_j)$ para cada d_j, y la

siguiente es una solución a r:

$$\{x \mapsto min_D(x), x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$$

- otra asignación de números reales, $d'_1, d'_2, \ldots, d'_k$, para el resto de las variables de la restricción, $x_1, x_2, \ldots, x_k$, tal que $min_D(x_j) \leq d'_j \leq max_D(x_j)$ para cada d'_j, y la siguiente es una solución a r:

$$\{x \mapsto max_D(x), x_1 \mapsto d'_1, \ldots, x_k \mapsto d'_k\}$$

Un PSR con varias restricciones es *consistente de frontera* si cada restricción c_i lo es a su vez. El permitir una asignación de números reales viene justificada por la posibilidad de encontrar fácilmente una asignación para el resto de las variables, basada en la definición de la restricción aritmética. La existencia de estos valores reales que están dentro del rango, aunque es una condición menos fuerte, puede verificarse más fácilmente que tratar de encontrar valores enteros, que es un problema combinatorio.

La definición se refiere a una *restricción primitiva*. Estas son las restricciones para las que se tiene una manera de verificar la consistencia. Para ver esto en más detalle consideremos la siguiente restricción:

- $X = 2Y + 3Z$
- D(X) = [0.,6], D(Y) = [0.,2], D(Z) = [0.,4]

Consideremos una reescritura de la restricción: $3Z = X - 2Y$. Si Z toma su máximo valor, el lado izquierdo de la ecuación sería 12, pero el valor máximo que puede tomar $X - 2Y$ es $6 - 2x0 = 6$. Luego, siguiendo la definición, no puede existir una asignación para X y Y tal que para el mayor valor de Z, sea una solución a la restricción. Por lo tanto no es consistente de frontera. En cambio, si reducimos el dominio D(Z) = [0.,2], puede verificarse que se obtiene consistencia.

En general, dados los rangos de cada variable en un restricción primitiva podemos tener métodos para obtener eficientemente los nuevos rangos, de tal forma que la restricción sea consistente de frontera. Nos referiremos a estos métodos como *reglas de propagación*.

Para la restricción $X = Y + Z$, podemos ver las reescrituras poniendo cada variable en el lado izquierdo de la ecuación:

$$X = Y + Z, \quad Y = X - Z, \quad Z = X - Y$$

consistencia-de-frontera-suma(D: Dominios)

$X_{min} \leftarrow max\{min_D(X), min_D(Y) + min_D(Z)\}$
$X_{max} \leftarrow min\{max_D(X), max_D(Y) + max_D(Z)\}$
$\mathrm{D}(X) \leftarrow \{d_X \in D(X) | X_{min} \leq d_X \leq X_{max}\}$
$Y_{min} \leftarrow max\{min_D(Y), min_D(X) - max_D(Z)\}$
$Y_{max} \leftarrow min\{max_D(Y), max_D(X) + min_D(Z)\}$
$\mathrm{D}(Y) \leftarrow \{d_Y \in D(Y) | Y_{min} \leq d_Y \leq Y_{max}\}$
$Z_{min} \leftarrow max\{min_D(Z), min_D(X) - max_D(Y)\}$
$Z_{max} \leftarrow min\{max_D(Z), max_D(X) + min_D(Y)\}$
$\mathrm{D}(Z) \leftarrow \{d_Z \in D(Z) | Z_{min} \leq d_Z \leq Z_{max}\}$
retornar D

Figura IV.2: Algoritmo: *consistencia de frontera* en la restricción $X = Y + Z$

Razonando sobre los mínimos y máximos valores de cada lado derecho podemos ver que:

$$\begin{aligned} X &\geq min_D(Y) + min_D(Z), & X &\leq max_D(Y) + max_D(Z) \\ Y &\geq min_D(X) - max_D(Z), & Y &\leq max_D(X) + min_D(Z) \\ Z &\geq min_D(X) - max_D(Y), & Z &\leq max_D(X) + min_D(Y) \end{aligned}$$

Estas desigualdades pueden usarse para obtener reglas que permitan garantizar consistencia de frontera a partir de los rangos actuales, y actualizando los rangos de las variables de acuerdo a los lados de derechos de la ecuaciones. Un algoritmo que implementa las reglas de propagación para la restricción $X = Y + Z$ puede verse en la figura IV.2

En los lenguajes o librerías de programación con restricciones se definen muchas de las restricciones primitivas que les permiten expresar expresiones aritméticas arbitrarias, o incluso algunas no lineales. Aunque muchas veces las restricciones primitivas equivalentes propagan menos información. Para algunas restricciones primitivas basta con aplicar estas reglas de propagación una vez, mientras que otras necesitan de varias iteraciones. En la figura IV.3 puede verse una algoritmo que toma en cuenta esto y obtiene, de ser posible, consistencia de frontera para un conjunto de restricciones. Toma en cuenta también que sólo se necesita volver a verificar la consistencia de una restricción si las variables involucradas son modificadas.

Pueden también definirse otras formas de consistencia para restricciones especializadas. Ejemplos de esta, es la restricción `all-different(`$\{V_1, \ldots, V_n\}$`)` que significa que todas las variables $V_1, \ldots, V_n$ son distintas. La implementación que se encuentra en las librerías

consistencia-de-frontera(R: Restricciones, D: Dominios)
Sea R de la forma $r_1 \wedge r_2 \ldots r_n$
$R_0 \leftarrow \{r_1, r_2 \ldots r_n\}$
Mientras $R_0 \not\equiv \emptyset$
  Tomar $r \in R_0$
  $R_0 \leftarrow R_0/\{r\}$
  $D_1 \leftarrow$ consistencia-de-frontera-primitiva(r, D)
  si $D_1 \equiv \emptyset$ entonces
    retornar D_1
  Para $i = 1$ hasta n
    si existe $x \in vars(c_i)$ tal que $D_1(x) \neq D(x)$ entonces
      $R_0 \leftarrow R_0 \cup \{c_i\}$
  D$\leftarrow D_1$
retornar D

Figura IV.3: Algoritmo: *consistencia de frontera* en un PSR

propaga más información que la sola conjunción de restricciones binarias $V_1 \neq V_2 \wedge V_1 \neq V_2 \wedge \ldots \wedge V_{n-1} \neq V_n$.

IV.1.3. Mantenimiento de Consistencia de Arco

La consistencia de nodo y de arco fueron pensadas como *preprocesamiento*. Es decir, se aplicaban al principio de la búsqueda y luego se procedía con *backtraking*. Ha dado muy buenos resultados aplicar consistencia de nodo y de arco en *cada* nodo del árbol de búsqueda. Es decir, cada vez que se instancia una variable, o se divide su dominio, se utilizan las consistencias para *propagar* la información que se acaba de dar. Este algoritmo se conoce como *Mantenimiento de Consistencia de Arco,*. En la figura IV.4 puede verse un algoritmo de este tipo. Utiliza consistencia de nodo y arco. En cada nodo toma uno de los valores y continua la búsqueda con esa suposición.

IV.2. Sistemas y ejemplos

Los sistemas para programación con restricciones vienen como librerías para ser usadas en diferentes lenguajes, entre las que destaca ILog[65](http://www.ilog.com), para ser usada en C++. Hay también variedad de lenguajes, entre ellos los primeros, y que siguen siendo usados, son los lenguajes de *Programación Lógica con restricciones* (Conocidos como CLP, por *Constraint Logic Programming*)[38]. Puesto que la programación lógica

```
mantenimiento-de-consistencia(R: Restricciones, D: Dominios)
  D← consistencia-de-arco(R, D)
  D← consistencia-de-nodo(R, D) /* Solo fue descrito informalmente */
  si D ≡ ∅ entonces
    retornar D
  sino si D es una valuacion entonces
    si D satisface R entonces
      retornar D
    sino
      retornar falso
  elegir una variable X tal que |D(X)| ≤ 2
  para cada x ∈ D(X)
    D1 ← mantenimiento-de-consistencia(R ∧X = x, D)
    si D1 ≢ falso entonces
      retornar D
  retornar falso
```

Figura IV.4: Algoritmo: *Mantenimiento de Consistencia* en un PSR

provee naturalmente soporte para la solución por *backtracking*, al colocar restricciones antes y durante la búsqueda, cada cambio es propagado a través de las variables. Con lo que obtenemos un comportamiento similar al *Mantenimiento de Consistencia*. Hay también otros lenguajes como Mozart/Oz[80] (http://www.mozart-oz.org) que proveen una arquitectura que permite también aprovechar paralelismo. Choco[64] (http://www.choco-constraints.net) es un librería desarrollada en Claire que permite las facilidades de ese lenguaje para modelar fácilmente el problema que se quiere resolver.

En la familia de los CLPs hay también historia, que comienza con CHIP[24] en el que se elaboraban las restricciones en lenguaje C. Más adelante, ante la necesidad de proveer maneras de introducir diferentes tipos de restricciones por parte del usuario, se ofrece un lenguaje intermedio para describirlas en términos de las variables, los diferentes cambios que pueden ocurrirles y los nuevos cambios que la restricción dispara. Esto para el caso particular de las consistencias de frontera u otras formas generalizadas. Siguen en desarrollo CHIP, ECLIPSE[82] (http://www-icparc.doc.ic.ac.uk/eclipse/) (que provee múltiples extensiones a Prolog), y GNU Prolog[23], antes conocido como clp(fd). Muchos de ellos proveen una interfaz para comunicarse con otros lenguajes, lo que permiten utilizarlos como librerías.

A continuación presentaremos algunos ejemplos comúnmente usados en la comunidad para mostrar las sintaxis de sus lenguajes o librerías. Uno de los más famosos, usado en

```
    S E N D
+   M O R E
-----------
  M O N E Y
```

Figura IV.5: SEND+MORE=MONEY: un problema clásico de satisfacción de restricciones

```
sendmore1(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    alldifferent(Digits),
    S #/= 0,
    M #/= 0,
                    1000*S + 100*E + 10*N + D
                + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    labeling(Digits).
```

Figura IV.6: Formulación de SEND+MORE=MONEY en ECLIPSE

Propagación de Restricciones en AI, es el `SEND+MORE=MONEY`, en el cual hay que encontrar la asignación de un dígito diferente para cada letra de la expresión de la figura IV.5, de manera que la suma tenga sentido.

Al formular el problema se debe definir las variables para las cuales se desea encontrar la solución, las letras que contiene la expresión. Se especifica que cada letra debe representar un dígito distinto. Las letras `S` y `M` no pueden ser 0, y se da una expresión que usando el valor posicional de cada letra, las relaciona para formar la suma que quiere satisfacerse. Esta formulación (extraída de ECLIPSE) se encuentra en la figura IV.6.

La ultima línea indica que debe usarse backtracking hasta obtener una valuación (*labelling*) para las variables del problema. ECLIPSE retorna la asignación que puede verse en la figura IV.7.

Diferentes formulaciones de mayor o menor complicación pueden llevar a tener soluciones más eficientes. En muchos casos se añade restricciones o información redundante a sabiendas de que la definición minimal en la mayoría de los casos no propaga toda la información posible. Otra formulación para el `SEND+MORE=MONEY` es la de la figura IV.8.

En ésta se utilizan variables para manejar explícitamente los acarreos (*carriers*) que relacionan la sumas de los dígitos en la misma posición para las tres palabras.

```
[eclipse 3]: sendmore1([S,E,N,D,M,O,R,Y]).

S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2
```

Figura IV.7: Respuesta de ECLIPSE para el SEND+MORE=MONEY

```
sendmore2(Digits) :-
        Digits = [S,E,N,D,M,O,R,Y],
        Digits :: [0..9],
        Carries = [C1,C2,C3,C4],
        Carries :: [0..1],
        alldifferent(Digits),
        S #/= 0,
        M #/= 0,
        C1             #= M,
        C2 + S + M #= O + 10*C1,
        C3 + E + O #= N + 10*C2,
        C4 + N + R #= E + 10*C3,
             D + E #= Y + 10*C4,
        labeling(Carries),
        labeling(Digits).
```

Figura IV.8: Otra formulación de SEND+MORE=MONEY en ECLIPSE

Un problema comúnmente usado para comparar el rendimiento es N-Reinas, que consiste en colocar en un tablero de ajedrez de NxN posiciones, N reinas sin que se ataquen entre ellas de acuerdo a las reglas del juego. Hay variedad de formulaciones. Las más exitosas utilizan reglas redundantes, diferentes estrategias para seleccionar la variable a tratar (cual reina) y que valores intentar primero (posiciones en el tablero). Pueden también definirse restricciones para este problema en particular.

Yendo más allá de las N-Reinas conviene enumerar algunas consideraciones al modelar y solucionar un problema:

- Introducción de restricciones redundantes que propaguen más información.
- *Selección de variables*: Un criterio usado es encontrar una falla lo antes posible. Para ello se puede tomar la variable con el dominio más pequeño, con el rango de valores más pequeño, o con mayor cantidad de restricciones que la involucren.
- La *selección de valores* tiene menos impacto en la búsqueda en general, pero si cuando esta cerca de la solución. Por ello suele intentarse primero aquellos que sean parte de la solución más factiblemente.
- La *forma de consistencia* usada. Para los problemas aritméticos con desigualdades, la propagación hecha por la consistencia de frontera es equivalente la de arco, y es más eficiente.

Todas estas ideas deben evaluarse en cada caso, puesto que usar algunas optimizaciones o criterios de desición podría implicar una degradación en el rendimiento. En capítulos posteriores estudiaremos el problema de Planificación y usaremos la programación con restricciones para resolverlo, donde lo anterior será tremendamente relevante.

Capítulo V

PLANIFICACIÓN COMO RAMIFICACIÓN Y PODA

Mirando los caminos recorridos en *Planificación* clásica y paralela, y como pueden ser relevantes varias ideas que han dado éxito recientemente; mirando también las técnicas usadas en *Scheduling* y las prácticas usadas en *Planificación Temporal*, podemos desarrollar un esquema que pone juntas varias de estas ideas. Esto nos permite explorar primeramente el campo de la planificación paralela, pero con ideas que se pueden llevar con cierta facilidad a la planificación temporal.

Introduciremos una formulación mediante ramificación y poda que integra: planificación no direccional, cotas inferiores, y uso de restricciones. Esto permite también hacer más clara la relación entre diferentes aproximaciones y otros problemas de optimización combinatoria.

V.1. Definiciones preliminares

En el caso de la planificación paralela, un *plan* P es una secuencia de conjuntos de acciones: $A_0, A_1, ..., A_{n-1}$ paralelas que debe satisfacer las condiciones de la figura II.6. Supondremos que cada acción aparece sólo una vez en todo el plan, esto es,

$$a \in A_i \rightarrow a \notin (A_1 \cup A_2 \cup \cdots \cup A_{i-1} \cup A_{i+1} \cup \cdots \cup A_{n-1}$$

Esto se conoce como *plan canónico*. Podemos definir $t_P(a)$ como el instante de tiempo de la acción a en el plan P:

$$t_P(a) \stackrel{\text{def}}{=} i \quad \text{si} \quad a \in A_i \wedge A_i \in P \tag{V.1}$$

Diremos que la acción a_i *precede* a_j en P, denotado como $a_i \prec a_j$, si $t(a_i) < t(a_j)$, y que a_i y a_j son *paralelas* en P cuando $a_i \not\prec a_j$ y $a_j \not\prec a_i$. Un plan P es un *plan válido* si y sólo si las acciones mutex (ver Eq. II.8) no son paralelas en P, y para toda acción a_i sus

precondiciones son ciertas en el instante $t(a_i)$, en general siguiendo las condiciones para la planificación paralela definidas en la sección II.4. De nuevo, estamos interesados en hallar un plan con el menor tiempo de completación.

V.2. Ramificación disyuntiva para Mutex y Enlaces Causales

El esquema de ramificación para el Job Shop (figura III.2) puede ser modificado para resolver el problema de planificación en IA. El esquema que desarrollaremos será factible para solucionar teorías en las cuales *algunos planes óptimos son canónicos*. Para dominios como los *Bloques* todos los planes son canónicos, pero algunas instancias de *Logística* no tienen planes canónicos (ver descripción de estos dominios en la sección II.1.3). Las teorías consideradas en los problemas de Scheduling, donde cada tarea es puesta en el plan *exactamente* una vez, son canónicas.

El punto de partida en la definición del estado inicial σ_0. En el Job Shop, σ_0 es definido como un conjunto de restricciones de precedencia

$$T(t_{ij+1}) \geq D(t_{ij}) + T(t_{ij}) \tag{V.2}$$

para cada par de tareas sucesivas t_{ij} y t_{ij+1} en cada trabajo. En planificación, las tareas (acciones) no están ordenados *explícitamente* por ciertas restricciones de precedencia, sino que el orden esta *implícito* en las precondiciones de las acciones. Este orden implícito no es único y, de hecho, parte de la complejidad del problema es ir construyéndolo, como se mostró en POP (sección II.2.1). Dicho orden puede explicitarse por medio de ecuaciones similares a las que caracterizan los estimadores heurísticos h^m. Por ejemplo, la Eq. II.5 para h^1 puede ser reescrita como:

$$h^1(C) = \begin{cases} \text{mín}_{a \in O(p)}[1 + h^1(a)] & \text{si } C = \{p\}, \text{ sino} \\ \text{máx}_{p \in C}\, h^1(\{p\}) & \text{si } |C| > 1 \end{cases} \tag{V.3}$$

donde asumimos que contamos con acciones *Inicial* y *Final*, como en POP. Para mantener la consistencia asumiremos también, $h^1(Inicial) = 0$, y $Inicial \in O(p)$ si p es cierto en la situación inicial. Así, $h^1(a)$ es una cota inferior para el tiempo necesario para comenzar la ejecución de la acción a

$$h^1(a) \stackrel{\text{def}}{=} h^1(prec(a)) \tag{V.4}$$

Estas ecuaciones para cotas inferiores pueden modificarse para que sean similares a las restricciones de precedencia del Job-Shop (Eq. V.2) por medio de dos transformaciones. Primero, proyectamos las ecuaciones de acciones proyectando el lado izquierdo de la ecuación V.4 para obtener

$$h^1(a) = \max_{p \in prec(a)} \left\{ \min_{a' \in O(p)} [1 + h^1(a')] \right\} \tag{V.5}$$

En segundo lugar, expresaremos la ecuación de cotas inferiores resultante, $h^1(a)$, en términos de las *variables de instante de tiempo* $T(a)$, donde $T(a)$ es una variable que contiene el *instante de tiempo en la cual la acción a es ejecutada* en el plan, cuyo dominio es el conjunto de los enteros no negativos extendido con ∞. $T(a) = \infty$ significa que la acción a no es ejecutada en el plan, y asumiremos, para fines de consistencia, que $a' \prec a$ es cierto para todos los a' donde $T(a) = \infty$. La forma de las restricciones de precedencia sobre dos variables de instante de tiempo $< a, a' >$ será:

$$T(a) < T(a') \tag{V.6}$$

que garantiza que la acción a se ejecuta antes que la acción a'.

Reformulando la ecuación V.5 en términos de las variables de instante de tiempo $T(a)$, obtenemos el conjunto de *restricciones de instante de tiempo*. Estas son apenas una primera aproximación, antes de completar el esquema:

$$T(a) \geq \min_{a' \in O(p)} [1 + T(a')] \quad \text{para cada } p \in prec(a) \tag{V.7}$$

Estas restricciones son similares a las restricciones de precedencia (Eq. V.2) para el Job Shop excepto por la presencia del *min*. Las llamaremos *restricciones de precondición*. No es difícil ver que las restricciones de precedencia y de precondición tienen propiedades computacionales similares: son tratables (solucionables en tiempo polinomial) y las cotas inferiores obtenidas $h(a)$ para cada variable de instante de tiempo $T(a)$ definen una solución consistente. Además, pueden ser solucionadas por medio de variaciones simples de algoritmos de ruta más corta en un grafo (*shortest path*). Llamemos a las teorías que combinan las restricciones de precondición y de precedencia *STP extendidos* (ESTP). Debido a la inclusión de ∞ en el dominio de las variables de instante de tiempo, los ESTPs siempre son consistentes ya que la asignación $T(a) = \infty$ siempre es una solución. Estamos interesados, por supuesto, en los valores consistentes más bajos para dichas variables. Especialmente para la variable $T(End)$ que es una cota inferior para el estado; *i.e.* $f(\sigma) = h_\sigma(End)$, donde $f(\sigma)$ es la cota inferior para un esquema de ramificación y poda (ver capítulo III).

Por la presencia de las eliminaciones en la formulación STRIPS, las ecuaciones definidas no son aún un esquema ni correcto ni completo, ya que las precondiciones de una acción pueden ser borradas antes de que la acción sea ejecutada. En POP (ver sección II.2.1) se usa la noción de *enlaces causales*[52], que permite detectar tales casos y corregirlos a fin de tener una solución correcta. Recordando, un enlace causal $a \xrightarrow{p} a'$ indica que la acción a precede a a', haciendo su precondición p verdadera, y pide que ninguna acción a'' que elimine p quede planificada para que se ejecute entre a y a'.

En este contexto usaremos los enlaces causales para obtener *mejores cotas inferiores.* A tal fin, introducimos las variables $S(p,a)$ para cada precondición p de una acción a y las llamaremos *variables de soporte.* Inicialmente, el dominio $D(p,a)$ de la variable de soporte $S(p,a)$ es $O(p)$, *i.e.*el conjunto de acciones que añade p. Sin embargo, $D(p,a)$ cambiará dinámicamente a medida que se intente un soporte en particular a' para el átomo p como precondición de a, $S(p,a) = a'$, lo que equivale a tener el enlace causal $a' \xrightarrow{p} a$; o cuando se descarte algún posible soporte porque ya se intentó y no se encontró solución, $S(p,a) \neq a'$.

Dada esta representación de los enlaces causales, las restricciones de precondiciones (Eq. V.7) pueden ser reescritas como:

$$T(a) > \min_{a' \in D(p,a)} T(a') \quad \text{para cada } p \in pre(a) \tag{V.8}$$

donde la minimización es hecha sobre el dominio *dinámico* $D(p,a)$ y no sobre que dominio *estático* $O(p)$. Note que una variable de soporte puede ser reducida de un conjunto inicial de valores($\{a_3, a_5, a_8, a_12\}$) a un sólo valor, solo indicando que todos los otros valores no forman parte del soporte ($S(p,a) \neq a_3 \wedge S(p,a) \neq a_8 \wedge S(p,a) \neq a_12$). Es así como se usa la información sobre las opciones que hemos rechazado, que es una característica de los enfoques de ramificación y poda. Cada vez que se elimine un soporte, esta regla podría propagar información.

La noción de enlaces causales en términos de variables de instante de tiempo y variables de soporte, se expresa:

$$T(a) \neq \infty \;\&\; S(p,a) = a' \;\&\; p \in del(a'') \;\Rightarrow\; a'' \prec a' \;\vee\; a \prec a'' \tag{V.9}$$

Este muestra una disyunción que debe ser satisfecha (no permitir que a'' interfiera en el enlace causal) cuando una condición se cumple (es posible que a'' interfiera con un enlace causal). Una ecuación indicando una disyunción bajo condición aparece también para los mutex (Eq. II.8):

$$mutex(a,a') \Longrightarrow a \prec a' \;\vee\; a' \prec a' \tag{V.10}$$

El esquema de ramificación y poda para *planificación canónica* (figura V.1) sigue del esquema para el Job-Shop (figura III.2) debido a las similaridades entre las planificaciones factibles del Job Shop, y los planes en IA.

Representemos una asignación sobre variables $T(a)$ por una lista de pares $P = \langle a_i, t_i \rangle_i$ tal que $T(a_k) = t_k \neq \infty$ si $a_i \in P$, y $T(a_k) = \infty$ si $a_k \notin P$. Luego, la tarea de planificación puede ser expresada con un *problema de satisfacción de restricciones* como sigue:

*$P = \langle a_i, t_i \rangle_i$ es un **plan canónico** válido si y sólo si la asignación P sobre las variables de instante de tiempo $T(a)$ y **algunas** asignaciones sobre las variables de soporte $S(p, a)$ satisfacen conjuntamente 1) las restricciones de precondición (Eq. V.8), 2) las restricciones de enlace causal (Eq. V.9), y 3) las restricciones de mutex (Eq. V.10)*

Yendo del esquema para el Job Shop (figura III.2) reemplazamos los estados de STPs por un par $\sigma = \langle Precs, Doms \rangle$. $Precs$ contiene todas las restricciones de precedencia con lo que obtenemos un ESTP, y Dom contiene las variables de instante de tiempo y de soporte, con sus respectivos dominios, que ya se introdujeron. Los estados terminales son *estados meta* si no tienen ninguna inconsistencia (bien sea por un enlace causal o algún mutex). Los hijos son generados seleccionando un conflicto a resolver.

Las restricciones de mutex y enlace causal son *disjuntivas* y al contrario de las restricciones de precondición y precedencia, son todas *intratables* computacionalmente. Como resultado, el esquema de ramificación y poda computa planificaciones relajadas h_σ considerando sólo las restricciones de precondición y las restricciones de precedencia que se hayan añadido, y ramifica sobre las disyunciones de mutex o de restricciones de enlace causal que están siendo violadas en h_σ. A estas las llamamos conflictos de enlace causal o mutex, y son similares a los conflictos en POP. El esquema resultante puede verse en la figura V.1.

V.3. Ramificación Guiada por la Meta

En los planificadores de orden parcial, y en otros como IxTex y RAX, el conjunto de acciones $Pasos$ a ser ordenados en el tiempo se construye incrementalmente, comenzado con las acciones $Inicio$ y $Final$, y verificando los conflictos solo con $Pasos$. Esto hace la búsqueda mas *guiada por la meta*, algo que a veces paga cuando en conjunto de acciones relevantes es pequeño en comparación con el conjunto de acciones disponibles. Esta modificación puede hacerse fácilmente en el esquema anterior resultando en lo que se muestra en la figura V.2.

1. Los estados $\sigma = \langle Precs, Doms \rangle$, donde $Precs$ son las restricciones de precedencia y precondición, y $Doms$ son los dominios de las variables de instante de tiempo, y las variables de soporte $T(a)$ y $S(p, a)$
2. El estado inicial $\sigma_0 = \langle Precs_0, Doms_0 \rangle$, donde $Precs_0$ son las restricciones de precondición (Eq. V.8) y $Doms_0$ son los dominios iniciales para las variables de instante de tiempo, y las variables de soporte
3. La planificación relajada h_σ se calcula resolviendo el ESTP $Precs$; y tomando como cota inferior $f(\sigma) = h_\sigma(End)$
4. Los estados terminales $\sigma = \langle Precs, Doms \rangle$ son aquellos donde ningún par de acciones (a, a') viola la restricción mutex (Eq. V.10), y ninguna tripleta (a, a', a'') viola la restricción de enlace causal (Eq. V.9) en la planificación relajada h_σ
5. Los hijos son generados de un estado no terminal $\sigma = \langle Prec, Doms \rangle$ seleccionando un conflicto mutex (a, a') y ramificando $[a \prec a'; a' \prec a]$ [a]; o para un conflicto de enlace causal (a, a', a''), ramificar en las siguientes opciones: $[S(p, a) \neq a'; S(p, a) = a', a'' \prec a; S(p, a) = a', a' \prec a'']$ (actualizando $Precs$ y $Doms$ apropiadamente).

[a] $[restriccion_1; restriccion_2]$ significa que tomamos primero una rama con $restriccion_1$, luego una con $restriccion_2$

Figura V.1: Esquema de Ramificación y Poda para Planificación Canónica

1. Los estados $\sigma = \langle Pasos, Precs, Doms \rangle$, **donde** $Pasos$ **es un conjunto de acciones**, y $Precs$ y $Doms$ como antes
2. El estado inicial $\sigma_0 = \langle \{Start, End\}, Precs_0, Doms_0 \rangle$ con $Prec_0$ y $Doms_0$ como antes
3. La planificación relajada h_σ se calcula resolviendo el ESTP $Precs$; y tomando como cota inferior $f(\sigma) = h_\sigma(End)$
4. Los estados terminales $\sigma = \langle Pasos, Precs, Doms \rangle$, son aquellos donde $|D(p,a)| = 1$ **para todo** $a \in Pasos$, ningún par de acciones (a, a') viola la restricción mutex (Eq. V.10), y ninguna tripleta (a, a', a'') viola la restricción de enlace causal (Eq. V.9) en la planificación relajada h_σ, con $a, a', a'' \in Pasos$.
5. Los hijos son generados de un estado no terminal $\sigma = \langle Pasos, Prec, Doms \rangle$ seleccionando un conflicto mutex (a, a') y ramificando $[a \prec a'; a' \prec a]$ o para un conflicto de enlace causal (a, a', a''), ramificar en las siguientes opciones: $[a'' \prec a; a' \prec a'']$; o **seleccionando un dominio** $|D(p,a)| > 1$ **para** $a \in Pasos$ **y una acción** $a' \in D(p,a)$ **y ramificando** $[S(p,a) = a'; S(p,a) \neq a']$. (actualizando $Precs$ y $Doms$ apropiadamente).

Figura V.2: Esquema de Ramificación y Poda Guiada por la Meta para Planificación Canónica

Capítulo VI

IMPLEMENTACIÓN

Un esquema de ramificación y poda como en la figura V.2, donde cada ramificación propaga información de las restricciones anteriores y las que se acaban de colocar, puede ser implementado usando *programación con restricciones* (capítulo IV). En particular puede ser adecuado un algoritmo de *mantenimiento de consistencia*, como el visto en la sección IV.1.3, donde en cada nodo propagamos información de acuerdo a las nuevas restricciones introducidas. Reportaremos la implementación del planificador BBP (por las siglas de *Branch-and-Bound Planner*, en castellano Planificador por Ramificación y Poda), basado en el esquema de la figura V.2. Este tiene semejanza con los planificadores de orden parcial (sección II.2.1), aunque construye planes paralelos óptimos y usa el estimador heurístico h^1 en forma de restricciones de precondición (Eq. V.8), aunque usando h^2_p (Eq. II.9), como cota inferior inicial del momento de inicio de las acciones, entre otras diferencias.

Esta formulación fue implementada en C++, usando como base el código de HSP 2.0[7] (ver sección II.3), y usando el lenguaje lógico de programación con restricciones GNU Prolog [23]. Se eligió un sistema de PR puesto que las decisiones de ramificación de nuestro algoritmo son *restricciones* que se agregan, y estos lenguajes las proveen y propagan sus efectos en forma *incremental* sobre las variables del problema. En nuestro caso, las variables de soporte $S(p, a)$ y de instante de tiempo para las acciones $T(a)$. GNU Prolog, en particular, siendo de dominio público, cuenta con una buena interfaz hacia C, y buen rendimiento[27].

VI.1. Algoritmo Básico

Una implementación de la ramificación y poda involucra el cálculo de la cota inferior del problema. Si se supera cierto límite, entonces no se continúa la búsqueda en esa dirección. Ésto equivale, en nuestro caso, al cálculo de la cota inferior $f(\sigma)$ para cada plan parcial σ, que conllevaría a verificar cada vez que $f(\sigma) \leq B$ para un límite B.

En la PR es común propagar las restricciones y verificar cada vez la consistencia del problema, y no una restricción en particular. El que la meta debe ser establecida en un instante de tiempo menor o igual a B se expresa con la restricción $T(Final) \leq B$. Si la incluimos dentro de las que tenemos inicialmente, podemos buscar la solución dentro de un horizonte limitado, mientras mantengamos la consistencia del problema. Para encontrar una solución óptima basta con partir desde un valor B dado por una cota inferior, e ir aumentando en uno la cota B hasta llegar a la solución. Esta longitud mínima es la cota inferior de la variable $T(Final)$ en el estado inicial σ_0. Un esquema del algoritmo resultante puede verse en la figura VI.1.

```
bbp()
  Establecer restricciones iniciales (precondicion, etc)
  B = cota inferior(Estado Final)
  Repetir
    Estado inicial = restricciones de precedencia + T(Final) ≤ B
    Si lazo-bbp(Estado inicial)
      reportar solucion y salir
    Sino
      B ← B +1

lazo-bbp(Estado)
  Si no existe conflicto por resolver
    retornar cierto
  Elegir un conflicto por resolver
  Repetir mientras existan opciones para resolver el conflicto
    Estado-hijo = elegir una opcion agregando una restriccion a Estado
    Si lazo-bbp(Estado-hijo)
      retornar cierto
    Sino
      descartar opcion asociada de Estado hijo
  retornar falso
```

Figura VI.1: Esquema del Algoritmo de BBP

La ramificación consiste en (1) elegir un conflicto a resolver e (2) ir explorando las posibles ramas asociadas a cada una de las opciones que se tienen para resolver dicho conflicto. Esto, hasta que se llegue a un estado final, en el que no hay conflicto, o se encuentre una inconsistencia, en cuyo caso se retorna con falla a fin de revisar las otras opciones disponibles. Nótese que estamos llamando *conflicto* a cualquier situación que impide que el plan sea correcto: un átomo por soportar, o interferencia entre acciones

(mutex o interferencia en enlace causal), como se indica en la figura V.2. Cada vez que se coloca una restricción, se propaga la información sobre las variables, siguiendo la idea del *mantenimiento de la consistencia* de PR. La idea de tener restricciones sobre el problema e ir resolviendo conflictos se utiliza otros problemas de optimización combinatoria [12] y en otros enfoques para la planificación, como en SATPLAN o CPLAN.

En este caso, estamos partiendo de las restricciones de precondición (Eq. V.8), que captura la heurística h_1. Durante la búsqueda se van añadiendo restricciones de precedencia (Eq. V.6) sobre las variables de instante de tiempo, que permiten ir construyendo un orden parcial que conlleva a un plan correcto. Y se añaden restricciones de igualdad y desigualdad sobre las variables de soporte, $S(p,a) = a'$ y $S(p,a) \neq a'$, que indican cuando se intenta o descarta -respectivamente- una acción a' entre los posibles soportes para una precondición p que debe satisfacerse. Los estados σ de la búsqueda, similares a los planes parciales, son entonces los dominios de las variables de instante de tiempo, y de las variables de soporte, además de las restricciones que se van estableciendo sobre ellas, siendo que en σ se mantienen las precedencias entre las acciones para que no interfieran entre sí.

Cuando se formulan problemas en términos de programación con restricciones, el encontrar la solución suele hacerse por un algoritmo de búsqueda que va eligiendo las variables a resolver, y el orden en que se le van asignando los diferentes valores posibles. El criterio utilizado para la elección de variables y ordenamiento de valores (ver las *heurísticas de selección de variables y de valores* en el capítulo IV) es de tremenda importancia. En general, debe decidirse que punto de decisión se resuelve primero, y cuál de las opciones disponibles se toma en ese caso. Un criterio general es adelantar las decisiones que lleven lo más pronto posible a saber si estamos en un estado inconsistente o no. El postergarlas haría crecer el árbol de búsqueda inútilmente.

Los conflictos por mutex o interferencia en enlace causal, se resuelven introduciendo precedencias entre estas acciones, decidiendo entre sólo dos ordenes posibles. En cambio, la elección de soportes para precondiciones involucra hasta cientos de opciones, muchas de las cuales son acciones que aún no forman parte del plan. Es preferible resolver primero los conflictos por mutex o interferencia y dejar de último la elección de soportes.

En resumen, los conflictos que se pueden presentar con sus correspondientes soluciones son:

Conflicto	**Opción 1**	**Opción 2**
a'' interfiere con $a \xrightarrow{p} a'$	$a'' \prec a$	$a' \prec a''$
mutex(a, a')	$a \prec a'$	$a' \prec a$
Atomo por soportar $S(p,a)$	Tomar uno: $S(p,a) = a'$	Descartarlo: $S(p,a) \neq a'$

El plan paralelo resultante corresponde a ejecutar cada acción lo antes posible, es decir, para la acción a, se ejecutaría en el instante $min(T(a))$. A fin de no procesar muchos conflictos potenciales, se resuelven sólo si significarían una inconsistencia en el plan paralelo resultante.

El resultado de refinar la función *lazo-bbp* a fin de reflejar estas ideas está en la figura VI.3, siguiendo la forma de las restricciones del esquema de ramificación y poda de la figura V.2.

Cada vez que se llama a *lazo-bbp* con una restricción, significa que esta se coloca antes de llamar a la función, y que en caso de que retorne falso se quitará. Nótese que agregar $S(p, a) = a'$ es equivalente a colocar el enlace causal $a' \xrightarrow{p} a$.

VI.2. Ejemplo

- Situación inicial: $\{i\}$
- Situación final: $\{f\}$
- Operadores:
  - Inicial: Prec $= \emptyset$, Add $= \{i\}$, Elim $= \emptyset$
  - Final: Prec $= \{f\}$, Add $= \emptyset$, Elim $= \emptyset$
  - a: Prec $= \{i\}$, Add $= \{o, q\}$, Elim $= \{p\}$
  - b: Prec $= \{i\}$, Add $= \{p\}$, Elim $= \emptyset$
  - c: Prec $= \{i\}$, Add $= \{r\}$, Elim $= \{q\}$
  - d: Prec $= \{o, p, r\}$, Add $= \{f\}$, Elim $= \emptyset$

Figura VI.2: Ejemplo donde aplican conflictos de enlace causal y mutex

A fin de observar en acción un posible comportamiento del algoritmo, partamos del ejemplo en la figura VI.2. No hay decisiones que tomar en cuanto a los soportes para las diferentes variables, puesto que para cada átomo hay sólo una opción; aunque el esquema que estamos siguiendo es *guiado por la meta* por lo que eso se constata sólo cuando se necesita soportar un átomo. Quedarían por resolver los conflictos por mutex y enlace causal.

Con un horizonte de búsqueda $B = 4$, el estado inicial para realizar las búsqueda asociada a este problema lo vemos en el cuadro VI.1.

Dominios de acciones	Soportes abiertos	Precedencias
$T(Inicial) = 0$ $T(a) = \{0..\infty\}$ $T(b) = \{0..\infty\}$ $T(c) = \{0..\infty\}$ $T(d) = \{0..\infty\}$ $T(Final) = \{0.,4\}$	$S(i, Final)$	$Inicial \prec Final$ $T(Final) \leq 4$

Cuadro VI.1: Estado inicial para ejemplo de la figura VI.2

lazo-bbp(Estado)
Si no existe conflicto por resolver
retornar *cierto*

//Tratar conflictos con enlaces causales.
Para cada accion a'' y enlace causal $a \xrightarrow{p} a'$, tal que $p \in Elim(a'')$
Si $min(T(a)) \leq min(T(a'')) \leq min(T(a'))$ // conflicto en el plan resultante
Si lazo-bbp(Estado-hijo) $a'' \prec a$
retornar *cierto*
Sino // unica opcion disponible
retornar lazo-bbp(Estado-hijo) con $a' \prec a''$

//Tratar mutex
Para cada par de acciones mutex a, a'
Si $min(T(a)) = min(T(a'))$
Si lazo-bbp(Estado-hijo) con $a \prec a'$
retornar *cierto*
Sino // unica opcion disponible
retornar lazo-bbp(Estado-hijo) con $a' \prec a$

Elegir un atomo por soportar $S(p, a)$
Repetir mientras $|D(p, a)| > 0$
Elegir $a' \in D(p, a)$
Si lazo-bbp(Estado-hijo) con $S(p, a) = a' \wedge a' \prec a$ y
agregando $p \in prec(a')$ a atomos por soportar, sino fue agregada antes
retornar *cierto*
Sino
$S(p, a) \neq a'$
retornar *falso*

Figura VI.3: Detalle de función *lazo-bbp*

Si resolvemos primero todos los soportes pendientes, obviando por simplicidad el que nuestro algoritmo es dirigido por la meta, agregaríamos los siguientes enlaces causales,

$$\begin{array}{ccccc} & & d \xrightarrow{f} Final & & \\ a \xrightarrow{o} d & & b \xrightarrow{p} d & & c \xrightarrow{r} d \\ Inicial \xrightarrow{i} a & & Inicial \xrightarrow{i} b & & Inicial \xrightarrow{i} c \end{array}$$

quedando el estado del cuadro VI.2.

Dominios de acciones	**Soportes abiertos**	**Precedencias**
		$Inicial \prec Final$
$T(Inicial) = 0$		$T(Final) \leq 4$
$T(a) = \{1.,2\}$		$d \prec Final$
$T(b) = \{1.,2\}$		$a \prec d$
$T(c) = \{1.,2\}$		$b \prec d$
$T(d) = \{2.,3\}$		$c \prec d$
$T(Final) = \{3.,4\}$		$Inicial \prec a$
		$Inicial \prec b$
		$Inicial \prec c$

Cuadro VI.2: Estado luego de resolver todos los soportes restantes del ejemplo de la figura VI.2

Mirando ahora los conflictos por interferencia en enlace causal, vemos que a interfiere con $b \xrightarrow{p} d$, puesto que elimina p, y que $min(T(b)) \leq min(T(a)) \leq min(T(d))$, por lo que el conflicto quedaría en el plan resultante. Para resolver este conflicto se presentan dos opciones: $a \prec b$ ó $d \prec a$. Pero intentar ésta última fallaría porque $a \prec d$ forma parte de las restricciones actuales. Lo que nos obliga a resolver con $a \prec b$. El estado resultante lo vemos en el cuadro VI.3.

Dominios de acciones	**Soportes abiertos**	**Precedencias**
		$Inicial \prec Final$
		$T(Final) \leq 4$
$T(Inicial) = 0$		$d \prec Final$
$T(a) = 1$		$a \prec d$
$T(b) = 2$		$b \prec d$
$T(c) = \{1.,2\}$		$c \prec d$
$T(d) = 3$		$Inicial \prec a$
$T(Final) = 4$		$Inicial \prec b$
		$Inicial \prec c$
		$a \prec b$

Cuadro VI.3: Estado luego de resolver un conflicto de enlace causal en el ejemplo de la figura VI.2

Dominios de acciones	Soportes abiertos	Precedencias
		$Inicial \prec Final$
		$T(Final) \leq 4$
$T(Inicial) = 0$		$d \prec Final$
$T(a) = 1$		$a \prec d$
$T(b) = 2$		$b \prec d$
$T(c) = 3$		$c \prec d$
$T(d) = 4$		$Inicial \prec a$
$T(Final) = 5$		$Inicial \prec b$
		$Inicial \prec c$
		$a \prec b$
		$a \prec c$

Cuadro VI.4: Estado luego de resolver un conflicto de enlace causal en el ejemplo de la figura VI.2

Igualmente $min(T(a)) = min(T(c))$, pero ellos son *mutex* puesto que c elimina q que es añadido por a. De las dos opciones, $a \prec c$ ó $c \prec a$, esta última nos diría que $Inicial \prec c \prec a \prec b \prec d \prec Final$. Pero esto es contradictorio con $T(Final) \leq 4$. Nos queda como estado sin conflictos el que vemos en el cuadro VI.4. Mirando los mínimos para las variables de instante de tiempo asociadas a las acciones, el plan paralelo resultante es comenzar con la acción a, luego b y c en paralelo, y finalmente d, es decir:

1. a
2. b, c
3. d

VI.3. Una disgresión: cálculo incremental del STP

En el esquema de ramificación y poda de la figura V.2 se usa el *Problema Temporal Simple* (STP, sección III.3). Durante la implementación de BBP, antes de codificar las diferentes heurística, calculábamos el STP procedente de las restricciones de precedencia introducidas durante las búsqueda. Primero, siguiendo el algoritmo original propuesto por *Dechter, et al*[22], que utiliza un algoritmo para calcular la ruta más corta en grafos[16]. Mediante este se calcula la ruta menor entre todos los nodos del grafo inducido a partir de las desigualdades (ver el artículo citado para más detalles). Pero este no es muy adecuado puesto que en cada nodo implica resolver el problema de nuevo. *Cesta, et al*[13] y *Gerevini,*

et al[33] presentan algoritmos que permiten actualizar el STP en presencia de una nueva desigualdad. Codificamos dichos algoritmos incrementales con resultados prometedores, lográndose solucionar en veinte minutos la instancia `log.a` del problema de logística, por ejemplo. Sin embargo, en la medida en que se complicaron las restricciones requeridas y a fin de gozar de una mayor flexibilidad se optó por el uso de un lenguaje con soporte para programación con restricciones. Este resultado parcial muestra, sin embargo, la relevancia de lo que se conocen como *algoritmos incrementales en grafos*[68], en el cálculo incremental de restricciones.

VI.4. Mejoras

Este esquema básico fue objeto de varias mejoras, que en algunos casos coinciden con las utilizadas en otros enfoques y planificadores. Todas ellas tendientes a adelantar la detección de caminos fallidos y así poder resolver en menor tiempo problemas de mayor complejidad.

VI.4.1. Cotas inferiores

El uso de h^1_{max} en la forma de restricciones de precondición lleva a cotas inferiores muy pobres. Una alternativa sería codificar el caso de h^m para $m = 2$ en el caso paralelo: h^2_p (Eq. II.9). Pero esto involucraría varias decenas de miles de restricciones y variables, y no es manejable por ninguna de las herramientas para programación con restricciones de las que disponíamos. De manera que hicimos un compromiso y usamos este estimador para calcular los valores en el estado *inicial*, a fin de reducir los dominios iniciales de las variables de acción $T(a)$. Es decir, al comenzar el problema se cumple que $T(a) \geq h^2_p(a)$ para cada acción a.

VI.4.2. Disyunciones activas

En nuestro esquema, las restricciones que tienen forma disyuntiva, la de mutex (Eq. V.10) y de enlace causal (Eq. V.9), juegan un rol pasivo en la búsqueda: se verifican en cada estado, y cuando son violadas en el plan relajado (las cotas inferiores de las variables $T(a)$), *una* de ellas es seleccionada para crear una rama y continuar la búsqueda en ese caso. En muchos trabajos [11, 3, 10] se intentan formas de inferencia limitada. Nuestras disyunciones son binarias, por lo que al verificar que una de ellas es falsa, necesariamente la otra tendrá que ser cierta y puede ser agregada. Esto evita ramificaciones y retornos (ver REPOP[59], donde se llega a conclusiones similares por otras razones). Por

ejemplo, para una disyunción $A \prec B \vee C \prec D$, pueden agregarse las siguientes reglas: $\neg(A \prec B) \rightarrow C \prec D$ y $\neg(C \prec D) \rightarrow A \prec B$. Actualmente esto se hace con todas las restricciones binarias.

Para que esto tenga mayor efecto, mantenemos actualizado el cálculo la clausura transitiva de todas las restricciones de precedencias agregadas, de manera que se pueda verificar la consistencia de una relación de precedencia en tiempo constante.

VI.4.3. Extensión de las eliminaciones por mutex implícitos

Siendo que la familia de heurísticas h^m da un estimado que no sobrestima la distancia real al objetivo, si $h^m(C) = \infty$ para un conjunto de átomos, entonces no existe ningún estado alcanzable desde el estado inicial en el cual aparezca el conjunto de átomos C. Tomando en cuenta la cota para la profundidad de la búsqueda B, en general no serán alcanzables los estados en los cuales aparezca el conjunto C si $h^m(C) > B$. Usando los valores calculados para h^2_p, diremos que los átomos p, q son mutuamente exclusivo (*mutex*) si $h^2_p(p, q) > B$. Este concepto está relacionado con los de GraphPlan que vimos en la sección II.4.1 [36]. Estamos usando también el nombre de mutex para hablar de los casos en que las acciones no pueden ejecutarse simultáneamente, pero esto no debería llevar a confusión, puesto que en todos los casos se refiere a la imposibilidad de que algunos atomos o acciones estén al mismo tiempo en el plan.

Una acción a *excluye* un átomo p cuando a elimina p o cuando a tiene como precondición o efecto positivo a q que es mutex con respecto a p. Puede mostrarse que las restricciones de mutex y enlace causal permanecen ciertas cuando las eliminaciones de cada acción a son reemplazadas por las eliminaciones extendidas, es decir, la lista de átomos p excluidos por a (ver RePop[59]).

VI.4.4. Extensión de conflictos de acciones a conflictos de variables de soporte

A fin de detectar conflictos adicionales de mutex y enlace causal, las variables de soporte $S(p, a)$, que representan la –posiblemente indeterminada– acción que soporta la precondición p de a, son tratamos como si fueran acciones normales.

Llamemos $T(p, a)$ a una variable de instante de tiempo asociada al momento en que se ejecuta la acción que se elija para soportar el átomo $p \in prec(a)$: $S(p, a)$. Puede verse que:

$$T(p, a) = \min_{a' \in D(p,a)} T(a') \tag{VI.1}$$

De manera que cuando a' es una acción en los *Pasos* del plan actual que elimina p, tal que $T(p,a) \prec a' \prec a$ es consistente, la disyunción $a' \prec T(p,a) \vee a \prec a'$ es agregada. Cuando a' es uno de los posibles valores de $S(p,a)$, debemos refinar la restricción. En este caso, la disyunción queda $S(p,a) = a' \ \vee \ (S(p,a) \neq a' \wedge (a' \prec T(p,a) \vee a \prec a'))$. Actualmente, sólo procesamos una de las reglas de la disyunción activa: $S(p,a) \neq a' \rightarrow (a' \prec T(p,a) \vee a \prec a')$. Además, las variables de instante de tiempo para los soportes también participan en la clausura transitiva de las precedencias, a fin de que las disjunciones también sean tratadas en forma activa.

VI.4.5. Coherencia con los conjuntos de soporte

Siguiendo a Weld[83], tratamos los conflictos incrementalmente. Es decir, cuando se agrega un enlace causal, un paso, o un átomo por soportar, se verifican contra los demás enlaces y pasos, generalizando también los conflictos a los conjuntos de soporte $S(p,a)$.

Usamos algunas restricciones adicionales a fin de mantener la consistencia entre las variables de soporte $S(p,a)$ y las variables de instante de tiempo asociadas $T(p,a)$ (Eq. VI.1). Podríamos reescribir la restricción de precondición Eq. V.8, como:

$$T(a) > T(p,a) \ \text{ para cada } p \in pre(a) \tag{VI.2}$$

Las dos siguientes muestran que cuando se sabe que una acción a' es el soporte de la acción a, entonces el instante de tiempo de a' y el soporte de a deben ser iguales:

$$S(p,a) = a' \ \rightarrow \ T(a') = T(p,a) \tag{VI.3}$$

y su contrapositiva:

$$T(a') \neq T(p,a) \ \rightarrow \ S(p,a) \neq a' \tag{VI.4}$$

Pero esta última conduce a poca propagación, por lo que combiene transformarlas en las siguientes, que además pueden aplicarse al colocar las restricciones de precedencia del antecedente.

$$a' \prec T(p,a) \quad \rightarrow \quad S(p,a) \neq a' \tag{VI.5}$$

$$T(p,a) \prec a' \quad \rightarrow \quad S(p,a) \neq a' \tag{VI.6}$$

También, al mostrar que una variable a' no puede ser mayor que una variable a, entonces no puede formar parte de los soportes de sus precondiciones:

$$\forall a' \in D(p,a) \;\; (min(T(a')) \geq max(T(a))) \;\rightarrow\; S(p,a) \neq a' \qquad \text{(VI.7)}$$

Las reglas anteriores mantienen la consistencia entre $S(p,a)$, $T(p,a)$ y las acciones, usando la idea general de que una acción a' sólo puede estar en un conjunto de soporte $S(p,a)$ si $a' = T(p,a)$ es consistente.

Varias restricciones pueden hacerse más estrictas si una acción excluye un átomo. Por ejemplo, si a' excluye una precondición de a, puesto que debe haber al menos un instante de tiempo entre ellas para que otra acción pueda restaurar esa precondición, la Eq. VI.7 quedaría como sigue:

$$\forall a' \in D(p,a) \;\; (min(T(a')) > max(T(a))) \;\rightarrow\; S(p,a) \neq a' \qquad \text{(VI.8)}$$

Igualmente, entre la variable tiempo para el soporte $T(p,a)$ y las acciones que forman parte del conjunto de soporte $S(p,a)$ tendrá que haber un instante de tiempo si esta excluye alguna precondición de a, por lo que su definición queda:

$$T(p,a) = \min_{a' \in D(p,a)} T(a') + \begin{cases} 1 & \text{si } a \text{ excluye } p' \in prec(a') \\ 0 & \text{si no} \end{cases} \qquad \text{(VI.9)}$$

A fin de sacar mayor provecho a todas estas reglas, optamos por agregar las restricciones correspondientes a todos los conflictos *potenciales*, aunque seguimos haciendo ramificación sólo en el caso de conflictos actuales, aquellos que harían que fuera incorrecto el plan que ejecuta las acciones lo antes posible.

VI.4.6. Heurísticas de selección

El planificador BBP incluye actualmente el algoritmo básico, las mejoras anteriores, junto con un criterio simple para seleccionar la variable de soporte $S(p,a)$, eligiendo primero aquella que tuviera menor cantidad de opciones $|D(p,a)|$, y rompiendo empates a favor de aquella cuya variable de tiempo asociada $T(p,a)$ tenga un menor rango, que es un criterio muy usado[79]. Éste criterio favorece el que se detectan las fallas más tempranamente. Exploramos otras combinaciones que incluían también la variable más restringida -de acuerdo al número de restricciones de precedencia en las que participaba-, la que propagara más reglas al introducir la precedencia, pero la combinación actual fue la que obtuvo mejor rendimiento en problemas conocidos utilizados como referencia por la comunidad.

De las acciones a soportar, se prefiere aquella que tuviera una cota superior más pequeña, rompiendo empates a favor de las acciones que ya formen parte del plan. Los estados que conducen a planes óptimos suelen tener las acciones restringidas, con pocas opciones

en el tiempo para ejecutarse. Esto, sumado al hecho de que el algoritmo sea *dirigido por la meta*, hace que esta heurística favorezca el hallazgo de una solución. Igualmente se probaron otras opciones basadas en rango, o número de restricciones en las que participa, pero esta fue la mejor opción.

Capítulo VII

EVALUACIÓN Y DISCUSIÓN

Nuestra implementación sigue siendo básica, aunque cumple el objetivo de estudiar la factibilidad del planificador como ramificación y poda, codificado mediante programación con restricciones. BBP es un planificador óptimo paralelo, que utiliza cotas inferiores para guiar una búsqueda no direccional. El cuadro VII.1 muestra el rendimiento de BBP, en relación a otros planificadores modernos, óptimos e independientes del dominio.

GraphPlan fue el primer planificador paralelo moderno (ver sección II.4.1). BlackBox es un planificador paralelo que combina las fortalezas de GraphPlan y SATPlan (ver sección II.4.3). Como SATPlan, podría usarse para obtener planes subóptimos, aunque en este caso sólo se usa en modo óptimo. IPP[43] es una mejora sobre la implementación original de GraphPlan.

hspr* (ver sección II.4.2) es un planificador óptimo que usa búsqueda heurística direccional. Puede usarse para obtener planes secuenciales y paralelos. En este caso lo usamos para obtener planes paralelos. Finalmente, TP4[37] es un planificador temporal que usamos para resolver los problemas paralelos. Su capacidad para lidiar con un problema de mayor complejidad hace que el rendimiento no sea directamente comparable con los demás, aunque lo reportamos por la relación entre la planificación paralela y temporal. No incluimos a Stan en la comparación puesto que este usa mecanismos adaptados especialmente para los dominios que usamos.

La implementación de GraphPlan que se usó, es la que viene implementada dentro de la distribución de BlackBox. Éste último fue usado con el solver para problemas proposicionales *Satz*, y usando la opción para compactar unitariamente la representación proposicional.

Las entradas en todas las tablas muestran tiempos medidos en segundos, ejecutados en una PC equipada con un procesador Intel Pentium III de 500 Mhz, con un máximo de 250MB por proceso. Para cada ejecución se permite un tiempo límite de 2000 segundos. Los símbolos '-T-' y '-M-' significan que el planificador agotó el tiempo o la memoria, respectivamente. "Longitud del plan" (*makespan*) se refiere al tiempo en el cual terminan

Problema	Longitud del plan	BBP	IPP	GraphPlan	BlackBox	hspr*	TP4
bw-large.a	12	3.04	0.15	0.35	4.8	0.22	0.92
bw-large.b	18	25.8	6.06	26.53	-M-	32	-T-
bw-large.c	28	GP	-M-	-M-	-M-	-T-	-T-
rocket.ext.a	7	40.63	16.36	-M-	2.67	85	184
rocket.ext.b	7	54.13	39.86	-M-	3.81	52	148
logistics.a	11	2.08	-M-	-M-	2.83	-T-	-T-
logistics.b	11	1094.97	-M-	-M-	10.65	-T-	-T-
logistics.c	13	NC	-M-	-M-	1743.7	-T-	-T-

Cuadro VII.1: Tiempo en cómputo del plan óptimo para varios planificadores paralelos y temporales modernos sobre un conjunto de problemas clásicos. BBP es el planificador propuesto.

las ultimas acciones del plan, es decir, el instante de tiempo en el cual se ejecutan las últimas acciones.

Como explicamos en los capítulos anteriores, los planes generados por BBP son canónicos. Es decir, que cada acción sólo puede aplicarse una vez en el plan. Muchos problemas pueden resolverse de esta manera, pero algunos no. En los resultados, GP indica que BBP tuvo problemas de memoria atribuibles a las limitaciones de GNU Prolog. NC significa que los planes óptimos encontrados con otros planificadores no son canónicos. En ese caso BBP intentaría buscar uno canónico óptimo, que puede no existir, o ser de igual o mayor longitud a los encontrados por otros planificadores.

Los problemas usados son ampliamente utilizados en la comunidad. Bloques y Logística fueron presentados en la sección II.1.3. "Bloques" consiste en unos bloques dispuestos en pilas sobre una mesa que son manipulados por los brazos de un robot. Incluye acciones para poner y quitar, de la mesa o encima de otro bloque. Existe otro conjunto de acciones para el dominio de bloques que permite explotar mayor paralelismo, pero que a muchos planificadores, incluyendo BBP, lleva a un uso excesivo de memoria o un grafo de búsqueda muy grande. Logística corresponde a un problema de transporte dentro de ciudades, y entre ellas, mediante camiones y aviones, respectivamente. Hay acciones para cargar, descargar y desplazar cada tipo de vehículo. Los problemas rocket pueden ser vistos como instancias de logística en las cuales sólo hay que transportar los objetos entre los aeropuertos. Suelen considerarse más sencillas que las de logística. Los problemas presentados no eran resueltos por ningún planificador de orden parcial, de manera que son referencia obligada desde la aparición de GraphPlan.

Algunos planificadores tienden a comportarse bien en dominios con altas interacciones, que conllevan a planes donde puede aprovecharse poco el paralelismo, como el problema

de bloques. GraphPlan, IPP, hspr* y TP4 se encuentran en esta categoría. Esto sucede porque tienen mecanismos para podar ramas fallidas, y finalmente realizan una búsqueda sobre un espacio muy reducido en complejidad. Otros manejan mejor los dominios con menos interacciones, aunque puedan tener muchos objetos presentes, como rocket y logística. Los planificadores de orden parcial, BlackBox, y otros planificadores basados en satisfacción o propagación de restricciones funcionan mejor en estos dominios. Esto lo logran tomando decisiones no direccionales en diversos puntos del plan. Sin embargo, en algunas ocasiones los enfoques basados en SAT o GraphPlan tienen problemas de memoria, puesto que representación para problemas de alta interacción tiende a ser muy grande.

En efecto, en el cuadro VII.1 vemos que BlackBox obtiene resultados muchos mejores en las instancias de logística, pero en las de bloques se ve superado o igualado por los otros planificadores. GraphPlan e IPPobtienen mejores resultados en bloques que en logística, donde se nota además la ventaja que da la implementación de IPP. hspr* y TP4 muestran su fortaleza en los bloques, al usar algoritmos direccionales, recordando que TP4 es un planificador temporal, lo que significa que su habilidad para lidiar con las duraciones diferentes de las acciones, afecta negativamente su rendimiento en los experimentos que hacemos. BBP mantiene un rendimiento intermedio entre los planificadores que manejan bien muchas interacciones, y dominios poco paralelos, como los bloques. Similar a GraphPlan e IPP en los bloques, y mejor que estos en logística, aunque no tanto como BlackBox.

Un dato importante es el número de nodos generados para conseguir la solución. Tiene que ver con cuanto del espacio de estados se logra podar por las restricciones y heurísticas. En el cuadro VII.2 puede verse este dato para los mismos problemas del cuadro VII.1, con los planificadores que lo reportan. *Rintanen*[70] reporta un planificador basado en la propagación con un conjunto de reglas obtenidas mediante observación del comportamiento de SATPlan y BlackBox. Éstos enfoques realizan gran cantidad de propagación sin búsqueda, aunque la gran cantidad de clausulas proposicionales restringen la usabilidad en muchos contextos. BBP también se haya en terreno medio, generando muchos menos nodos que los enfoques basados en búsquedas heurística, entre los que incluimos a GraphPlan, siguiendo a *Haslum y Geffner*[36], aunque generamos más que los reportados por *Rintanen*.

Decidimos probar en problemas más pequeños con los planificadores de rendimiento superior o similar, esta vez con un límite de tiempo de 300 segundos a fin de cubrir mayor cantidad de problemas. Se generaron instancias del problema de bloques usando la herramienta `bwstates`[76], y se reporta el rendimiento en el cuadro VII.3. Los problemas más sencillos favorecieron a IPP. BBP tuvo problemas en dos instancias (7-1 y 8-2)

Problema	BBP	IPP (act+nop)	HSPr*	TP4	*Rintanen* ver [70]
bw-large.a	85	2305	77	133	NR
bw-large.b	450	276787	28128	-T-	2
bw-large.c	GP	-M-	-T-	-T-	1573
rocket.ext.a	13131	1636829	238308	1629506	31
rocket.ext.b	17402	3050989	105366	1236001	26
logistics.a	322	-M-	-T-	-T-	14
logistics.b	212384	-M-	-T-	-T-	438
logistics.c	NC	-M-	-T-	-T-	3119

Cuadro VII.2: Número de nodos generados en el cómputo del plan óptimo para varios planificadores paralelos y temporales modernos sobre un conjunto de problemas clásicos. BBP es el planificador propuesto.

que fueron resueltas por BlackBox también, aunque BBP obtuvo en general mejor rendimiento. Estas faltas parecen indicar la necesidad de afinar las reglas de optimización, a fin de que capten algunos casos que faltan por detectar aún.

Del dominio `logística` se tomaron instancias extraídas de la primera ronda de la competencia de planificadores de *Artificial Intelligence Planning System* 2000, sin uso de tipos en la especificación PDDL. Los resultados se aprecian en el cuadro VII.4. Para los problemas 6-2, 6-9, 7-0 y 7-1, los planes encontrados resultaron ser no canónicos, por lo que los planes encontrados por BBP tenían como longitud del plan 11, 11, 13 y 13, respectivamente. BlackBox resolvió sin problemas todas las instancias. Esta vez no se observa superioridad de BBP sobre IPP, probablemente debido a la sencillez de las instancias.

Problema	Nro. Bloques	Longitud del plan	BBP	IPP	BlackBox
5-1	5	9	0.44	0.02	0.348
5-2	5	9	0.34	0.02	0.186
5-3	5	5	0.21	0.01	0.088
5-4	5	9	0.48	0.02	0.632
5-5	5	3	0.19	0.02	0.048
6-1	6	11	0.82	0.04	0.725
6-2	6	11	1.17	0.04	0.881
6-3	6	9	0.82	0.03	0.718
6-4	6	5	0.35	0.02	0.061
6-5	6	9	0.76	0.04	0.745
7-1	7	17	28.15	0.11	6.344
7-2	7	11	1.32	0.07	1.605
7-3	7	7	0.75	0.05	0.943
7-4	7	7	1.21	0.05	0.807
7-5	7	15	7.52	0.07	2.534
8-1	8	17	7.52	0.37	9.755
8-2	8	17	-T-	2.48	79.695
8-3	8	9	2.65	0.13	8.346
8-4	8	19	9.93	0.18	23.453
8-5	8	21	2.52	0.10	13.680

Cuadro VII.3: Tiempo de cómputo del plan óptimo para varios planificadores paralelos y temporales modernos sobre un conjunto de problemas de problemas de Bloques generados con `gen-bw`. BBP es el planificador propuesto.

Problema	Paquetes	Ciudades	Longitud del plan	BBP	IPP	BlackBox
4-0	4	2	9	0.29	0.04	0.384
4-1	4	2	9	0.27	0.04	0.447
4-2	4	2	9	0.25	0.03	0.413
5-0	5	2	9	0.33	0.04	0.422
5-1	5	2	9	0.29	0.04	0.416
5-2	5	2	3	0.25	0.02	0.08
6-0	6	2	9	0.34	0.04	0.438
6-1	6	2	9	0.26	0.03	0.483
6-2	6	2	9	0.59	0.04	0.417
6-9	6	2	11	0.40	0.04	0.901
7-0	7	3	12	2.96	6.46	1.286
7-1	7	3	13	1.91	110.66	1.983
8-0	8	3	11	0.97	1.41	1.002
8-1	8	3	12	1.99	5.71	1.288
9-0	9	3	11	0.79	0.33	0.783
9-1	9	3	10	86.83	0.25	2.13

Cuadro VII.4: Tiempo de cómputo del plan óptimo para varios planificadores paralelos y temporales modernos sobre un conjunto de problemas de problemas de logística tomados de la competencia de planificadores en AIPS 2000. BBP es el planificador propuesto.

Capítulo VIII

CONCLUSIONES Y TRABAJOS RELACIONADOS

La *planificación como búsqueda heurística* [53, 6, 7] está basada en esquemas de ramificación direccional. El planteamiento de ésta *como ramificación y poda* es una extensión a un esquema *no-direccional* que ha mostrado ser más adecuado en contextos paralelos y temporales. La principal contribución de este trabajo es mostrar la *factibilidad* de este esquema. Esto se logra planteándolo como un esquema de *ramificación y poda*, implementado por medio de *programación con restricciones* que permite integrar fácilmente enfoques que muchas veces se presentaron como alternativas excluyentes (SAT y CSP *versus* búsqueda heurística).

El rendimiento obtenido no iguala a los mejores planificadores paralelos óptimos actuales, basados en satisfacción y otras formas de propagación de restricciones. Sin embargo, los resultados son alentadores y ya han sido publicados[60]. Alcanza rendimientos similares o superiores a GraphPlan y a la planificación heurística paralela (hspr*); algo que aún no se tiene para planificadores óptimos de orden parcial (POP). Esto, junto con el hecho de que la implementación puede ser mejorada y extendida para manejar acciones de duración no unitaria, hace a esta aproximación interesante y susceptible a una exploración más profunda.

La *programación con restricciones* también permitió gran *flexibilidad*, y potencia la exploración e *integración* de otras técnicas, lo que potencia las posibilidades de desarrollo de planificadores más expresivos y eficientes. Nuevas optimizaciones pueden ser obtenidas e integradas de todos los trabajos relacionados con SATPlan, GraphPlan, y la planificación de orden parcial.

El uso de *heurísticas en planificación orden parcial* fue considerada recientemente en RePop[59] donde se presenta un planificador de orden parcial, sub-óptimo, y de alto rendimiento. Aunque nuestro planificador BBP fue desarrollado independientemente, hay algunas ideas en común entre ambos planificadores, como explotar la información de mutex implícitos (como en GraphPlan o usando la heurística h^2_p), y el razonamiento sobre

disyunciones. Rintanen en [70] explica las limitaciones de GRAPHPLAN y las ventajas de BLACKBOX en términos de un esquema de ramificación no direccional. Sus planteamientos son desarrollados en [50]. También está relacionado CPLAN[81] que es un planificador de alto rendimiento basado en restricciones que requieren de un modelo altamente entonado para cada dominio especifico. Si bien estamos atacando la planificación *independiente del dominio*, muchos de sus planteamientos son relevantes y tienen relación con este trabajo.

Una importante desventaja, como se discutía en el capítulo anterior, es el uso de h^1_{max} como estimador heurístico que asume que el tiempo necesario para lograr hacer cierto un conjunto de átomos, es dado por el tiempo necesario para lograr cada átomo *individualmente*. Esto es de suma importancia, puesto que los planificadores actuales podan mucho espacio de búsqueda detectando *interacciones* entre pares de átomos. BLACKBOX utiliza el esquema de satisfacer una traducción proposicional, al igual que SATPLAN pero codificando el grafo de GRAPHPLAN, que como se vio en la sección II.4.1, captura información equivalente a h^2_p[36], una cota mucho más informada. De hecho, HSPr*[36] la usa junto con búsqueda direccional IDA* y logra resultados importantes en planificación paralela óptima. TP4[37] hace planificación heurística temporal con una heurística similar.

El que este esquema pueda ser extendido de manera que acepte acciones con duraciones no unitarias, acerca los problemas históricamente diferenciados entre la inteligencia artificial, que lidian con la interacción de las acciones, y la investigación de operaciones, que se ocupan del orden de las acciones cuya interacción ya se conoce (en [77] se discute la relación entre ambos problemas). Algunos sistemas con que combinan la planificación, como se entiende en la IA, y la planificación de tareas de IO, incluyen a IXTEX [47] y RAX [40] (véase también [57, 26, 20, 86]) aunque tienen un rendimiento pobre cuando no se le añade información especifica del dominio, debido a que usan cotas inferiores muy bajas.

Capítulo IX

TRABAJOS FUTUROS

La más seria limitación en el planificador y su formulación subyacente es el que sólo considere planes canónicos, donde las acciones instanciadas son ejecutadas a lo más una vez. Una manera de superar esta limitación es reemplazar las restricciones de precondición dinámicas (Eq. V.8), que depende del dominio *actual* $D(p, a)$ de las variables de soporte $S(p, a)$, por restricciones estáticas de precondición (Eq. V.7)que dependen solamente de los dominios *iniciales*. De esta manera, un número arbitrario de 'copias' de las acciones pueden ser añadidas al plan, aunque esto resulte en una menor propagación con lo que se podrían tener cotas menores que afecten el rendimiento. Por otro lado, si uno sabe que las acciones pueden ser repetidas un número máximo pequeño de veces n, entonces es suficiente agregar $n-1$ copias $\{a^1, a^2, ..., a^{n-1}\}$, para cada acción a^0 en el dominio, junto con restricciones que garanticen que una acción a^j sólo puede estar en el plan, si esta la acción a^i con $i < j$. De esta manera, elimiríamos las simetrías que harían el problema mucho más complejo.

El problema de ir de planificación canónica a no-canónica sigue abierto. Sin embargo, hacer la planificación de manera que cada acción es ejecutada *a lo más* una vez tiene sentido desde el punto de vista de integrar el problema con otros de investigación de operaciones, donde generalmente las acciones son ejecutadas exactamente *una* vez.

Asumiendo esta limitación, existen tres maneras en las cuales creemos se puede mejorar la implementación actual. En primer lugar, es necesario dedicar más tiempo a afinar las heurísticas de selección de variables y valores, que usamos para seleccionar cual de las variables de precondición por soportar $S(p, a)$ debe atacarse primero, y el orden en que se considerarán los diferentes soportes $a' \in D(p, a)$. Estas heurísticas tienen impacto importante en el rendimiento.

Segundo, es necesario mejorar las reglas de propagación, de manera que propaguen más información. Actualmente, se generan unas pocas decenas de nodos por segundo, y aunque el número es con frecuencia menor al de otros planificadores, esto no necesariamente se

traduce en mejores tiempos.

En este momento las restricciones de precondición son usadas para capturar la heurística h^1. Es necesario extenderlas para que capturen cotas más informadas como h^2_p. El uso de esta última como restricción inicial sobre los dominios tiene gran importancia en haber logrado los resultados actuales. Si se pudiera además propagar esta información con las restricciones adicionales introducidas durante la ejecución, integraríamos aún más las mejores técnicas de los diferentes enfoques.

El esquema y las restricciones de BBP puede extenderse con relativa facilidad a la planificación temporal, aún más por el uso de programación con restricciones. En la reciente competencia de planificadores de AIPS 2002, se hizo especial énfasis en la planificación temporal. Muchas técnicas usadas allí pueden ser aplicadas con relativa facilidad.

Referencias Bibliográficas

[1] F. Bacchus. The 2000 AI Planning Systems Competition. *Artificial Intelligence Magazine*, 2001.

[2] E. Balas and P. Toth. Branch and bound methods. In E. L. Lawler *et al.*, editor, *The Traveling Salesman Problem*, pages 361–401. John Wiley and Sons, Essex, 1985.

[3] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proc. IJCAI-95*, 1995.

[4] A. Blum and M. Furst. Fast planning through planning graph analysis. In C. Mellish, editor, *Proceedings of IJCAI-95*, pages 1636–1642. Morgan Kaufmann, 1995.

[5] B. Bonet and H. Geffner. High-level planning and control with incomplete information using POMDPs. In G. Di Giacomo, editor, *Proceedings AAAI Fall Symp. on Cognitive Robotics*, pages 52–60. AAAI Press, 1998.

[6] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of ECP-99*, pages 359–371. Springer, 1999.

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

[8] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719. MIT Press, 1997.

[9] T. Bylander. The computational complexity of STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[10] J. Carlier and E. Pinson. An algorithm for solving the job shop scheduling problem. *Management Science*, 35(2), 1989.

[11] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proc. ICLP-94*, pages 369–383. MIT Press, 1994.

[12] Yves Caseau and Francois Laburthe. Solving small TSPs with constraints. In *International Conference on Logic Programming*, pages 316–330, 1997.

[13] R. Cervoni, A. Cesta, and A. Oddi. Managing dynamic temporal constraint networks. In *Proc. AIPS-94*, pages 13–20, 1999.

[14] A. Cesta, A. Oddi, and S. Smith. Profile based algorithms to solve multicapacited metric scheduling problems. In *Proc. AIPS-98*, 1998.

[15] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[16] Cherkassky, Goldberg, and Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1994.

[17] A. Cimatti and M. Roveri. Conformant planning via model checking. In *Proceedings of ECP-99*. Springer, 1999.

[18] William J. Cook and William H. Cunningham. *Combinatorial Optimization*. John Wiley and Sons, 1997.

[19] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[20] K. Currie and A. Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[21] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[22] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[23] D. Diaz and P. Codognet. Design and implementation of the GNU-Prolog system. *Journal of Functional and Logic Programming*, 2001. System at `http://pauillac.inria.fr/ diaz/gnu-prolog`.

[24] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS88)*, 1988.

[25] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second Int. Conference on Artificial Intelligence Planning Systems*, pages 31–36. AAAI Press, 1994.

[26] S. Chien *et al.* Aspen – automating space mission operations using automated planning and scheduling. In *Proc. SpaceOps2000*, Toulouse, France, 2000.

[27] A. Fernández and P. M. Hill. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Journal of Constraints*, 5:275–301, 2000.

[28] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

[29] M. Fox and D. Long. PDDL2.1 – an extension to pddl for expressing temporal planning domains. Available at `http://www.dur.ac.uk/d.p.long/IPC/pddl.html`, 2001.

[30] H. Geffner. Functional strips: a more general language for planning and problem solving. Logic-based AI Workshop, Washington D.C., 1999.

[31] H. Geffner. Planning as branch and bound and its relation to constraint-based approaches (longer version). Technical report, Universidad Simón Bolívar, 2001. Available at `www.ldc.usb.ve/~hector`.

[32] H. Geffner. Perspective on ai planning. In *Proceedings AAAI-2002*, 2002.

[33] A. Gerevini, A. Perini, and F. Ricci. Incremental algorithms for managing temporal constraints. Technical report, IRST, Italy, 1996. Tec. Rep. IRST-9605-07.

[34] Teofilo F. Gonzalez and Sartaj Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23(4):665–679, 1976.

[35] Durham Planning Group. What is planning? `http://www.dur.ac.uk/~dcs0www/research/stanstuff/ html/whatisplanning.html`, 2002.

[36] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proc. of the Fifth International Conference on AI Planning Systems (AIPS-2000)*, pages 70–82, 2000.

[37] P. Haslum and H. Geffner. Heuristic planning with time and resources. In *Proc. IJCAI-01 Workshop on Planning with Resources*, 2001.

[38] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1–3):113–159, 1992.

[39] Jörg Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems (ISMIS-00)*, pages 216–227. Springer, October 2000.

[40] A. Jonsson, P. Morris, N. Muscettla, and K. Rajan. Planning in interplanetary space: Theory and practice. In *Proc. AIPS-2000*, 2000.

[41] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, pages 1194–1201. AAAI Press / MIT Press, 1996.

[42] H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. In T. Dean, editor, *Proceedings IJCAI-99*, pages 318–327. Morgan Kaufmann, 1999.

[43] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. Proc. 4th European Conf. on Planning (ECP-97). Lect. Notes in AI 1348*, pages 273–285. Springer, 1997.

[44] R. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[45] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.

[46] R. Korf and L. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of AAAI-96*, pages 1202–1207. MIT Press, 1996.

[47] P. Laborie and M. Ghallab. Planning with sharable resources constraints. In C. Mellish, editor, *Proc. IJCAI-95*, pages 1643–1649. Morgan Kaufmann, 1995.

[48] E. Lawler and A. Rinnooy-Kan, editors. *The Traveling Salesman Problem : A Guided Tour of Combinatorial Optimization*. Wiley, 1985.

[49] D. Long and M. Fox. The efficient implementation of the plan-graph. *JAIR*, 10:85–115, 1999.

[50] S. Marcugini M. Baioletti and A. Milani. DPPlan: An algorithm for fast solution extraction from a planning graph. In *Proc. AIPS-2000*, 2000.

[51] K. Marriot and P. Stuckey. *Programming with Constraints*. MIT Press, 1999.

[52] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of AAAI-91*, pages 634–639, Anaheim, CA, 1991. AAAI Press.

[53] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proc. Third Int. Conf. on AI Planning Systems (AIPS-96)*, 1996.

[54] D. McDermott. PDDL – the planning domain definition language. Available at `http://ftp.cs.yale.edu/pub/mcdermott`, 1998.

[55] D. McDermott. The 1998 AI Planning Systems Competition. *Artificial Intelligence Magazine*, 21(2):35–56, 2000.

[56] G. Monahan. A survey of partially observable markov decision processes: Theory, models and algorithms. *Management Science*, 28(1):1–16, 1983.

[57] N. Muscettola. HSTS: Integrating planning and scheduling. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[58] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, 1980.

[59] Xuan Long Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *Proc. IJCAI-01*, 2001.

[60] Hector Palacios and Hector Geffner. Planning as branch and bound: A constraint programming implementation. In *XXVIII Latin American Conference on Computer Science - CLEI'02 - UNESCO*, Montevideo, Uruguay, November 2002. UNESCO. to appear.

[61] J. Pearl. *Heuristics*. Addison Wesley, 1983.