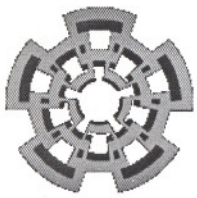

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL
UNIDAD ZACATENCO

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN COMPUTACIÓN

SISTEMAS DE PLANEACIÓN BASADOS EN EL ENFOQUE DE AUTÓMATAS FINITOS

TESIS PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS EN LA ESPECIALIDAD DE INGENIERÍA ELÉCTRICA OPCIÓN COMPUTACIÓN

Presenta:

Lic. Pedro Garduño Carmona

Director de tesis:

Dr. Guillermo Morales Luna

México D.F. marzo de 2002.

Introducción

El estudio de los sistema de planeación [13] en los últimos años ha sufrido muchos cambios. Existen diferentes maneras de resolver el problema de planeación [4]; en algunas se usan métodos heurísticos los cuales proporcionan una manera veloz de resolver el problema pero tienen el inconveniente de no siempre encontrar una solución; en otras se utilizan métodos basados en proposiciones como el *Graphplan* y compiladores que traducen el problema de planeación a problemas de proposición [5].

Esta tesis pretende solucionar el problema de planeación utilizando una técnica de autómatas finitos. La técnica de autómatas finitos pretende ser una técnica novedosa ya que a grandes rasgos el problema de planeación se traduce en un autómata finito y la teoría de autómatas finitos se encuentra muy estudiada y fácil de tratar. Para esto en el capítulo 1 se da una introducción de la teoría de los autómatas finitos [1], [2] con el propósito de tener un panorama general de la teoría de los autómatas finitos. Adicionalmente también se estudiarán gramáticas regulares, el teorema de Kleene [1] y la regla de Arden.

En el capítulo 2 estudiamos a los sistemas de planeación [4], [5] y algunos métodos utilizados para resolver el mismo problema, con el fin de tener un panorama general de los problemas de planeación y la manera en que se ataca el problema desde otro punto de vista. En este capítulo se estudia a detalle el método implementado por el `Graphplan` [4] y se hace una comparación

con el método utilizado en este documento. También presentamos de manera formal a los sistemas de planeación. En el capítulo 3 hablamos un poco de algunos métodos para resolver problemas en general [14] y tratamos a los árboles binarios [18], los cuales son muy utilizados por algunos métodos de resolución del problema. En el capítulo 4 tratamos el problema de planeación por medio de los autómatas finitos, apoyándonos mucho en lo visto en el capítulo 1 pues es ahí donde se sientan las bases para el diseño del algoritmo propuesto en este trabajo. En el último capítulo se presenta el diseño del sistema que se elaboró para poder apoyar lo antes visto en el capítulo 4 y el desarrollo del software utilizando el lenguaje de programación *Python* [20], [17]. En este capítulo también presentaremos un ejemplo completo en pseudo-código, a fin de entender mejor el funcionamiento del algoritmo. Este capítulo también incluye un diagrama de bloques del sistema y describe el funcionamiento general del sistema.

Índice general

Capítulo 1

Autómatas finitos

En este capítulo veremos algunos conceptos básicos sobre autómatas finitos, con la finalidad de sentar las bases para el entendimiento de los siguientes capítulos.

1.1. Definiciones

Un problema con el que se enfrenta un programador de compiladores consiste en decidir si una cadena es válida. Por ejemplo, una variable en el lenguaje C comienza típicamente con una letra, e.g.: $x23$, AbC. Un ejemplo de una cadena no válida como variable en lenguaje C sería: 23a. El problema radica en poder diseñar un algoritmo que reconozca ciertos patrones dependiendo del tipo que queremos reconocer.

Una *relación* (binaria) es un conjunto de pares. El primer componente de cada par se toma de un conjunto llamado *dominio* y el segundo componente de cada par pertenece a un conjunto llamado *contradominio*. Al principio sólo utilizaremos relaciones en las que el dominio y el contradominio están en el mismo conjunto S. En este caso diremos que la relación es *sobre* S. Si R es una relación y (a, b) es un par de R, entonces, con frecuencia escribiremos aRb.

Propiedades de las relaciones

Decimos que una relación R sobre S es

1. *reflexiva* si aRa para toda a en S es verdadera
2. *irreflexiva* si aRa es falsa para toda a en S
3. *transitiva* si aRb y bRc implican aRc
4. *simétrica* si aRb implica bRa
5. *asimétrica* si aRb implica que bRa es falsa

Notemos que cualquier relación asimétrica debe ser irreflexiva.

Ejemplo 1.1. La relación $<$ sobre el conjunto de los enteros es transitiva porque $a < b$ y $b < c$ implica que $a < c$. Es asimétrica, y por tanto irreflexiva, porque $a < b$ implica que $b < a$ es falsa.

Cuando se trata de analizar cadenas siempre es necesario saber qué ocurrencias de símbolos podrían aparecer. Por ejemplo, en lenguaje C, para definir una variable es posible tener las letras del alfabeto, números y algunos carácteres especiales. A ese conjunto de símbolos le llamaremos alfabeto y lo denotaremos por Σ.

Ejemplo 1.2. Ejemplos de alfabeto

1. $\Sigma = \{a, b, c, d, e, \ldots, z\}$
2. $\Sigma = \{0, 1\}$
3. $\Sigma = \{0, 1, 2\}$

Definición 1.1. *Un autómata finito determinista consiste en una quíntupla* $(S, \Sigma, \delta, i, F)$ *donde:*

1. *S es un conjunto finito de estados*

2. *Σ es el alfabeto de la máquina*

3. *δ es una función (llamada función de transición) de $S \times \Sigma$ a S*

4. *i (un elemento de S) es el estado inicial*

5. *F (un subconjunto de S) es el conjunto de estados de aceptación*

De aquí en adelante denotaremos a un autómata finito determinista por las iniciales *AFD*.

La interpretación de la función de transición δ de un *AFD* es $\delta(p, x) = q$ si y sólo si el autómata puede pasar de un estado p a un estado q cuando la función de transición utiliza el símbolo x. Ya que el dominio de la función δ es $S \times \Sigma$ ($\delta : S \times \Sigma \to S$).

Por lo tanto, el autómata finito $(S, \Sigma, \delta, i, F)$ acepta la cadena $x_1, x_2, \ldots, x_n$ si y sólo si existe un conjunto de estados $s_0, s_1, \ldots, s_n$ tal que $s_0 = i \in F$, y para cada entero $j = 1, \ldots, n$ $\delta(s_{j-1}, x_j) = s_j$.

Definición 1.2. *Sea $M = (S, \Sigma, \delta, i, F)$ un autómata finito, definimos $\delta^* : S \times \Sigma \to S$ de manera recursiva como sigue:*

$$\begin{aligned} \delta^*(p, nil) &= p \\ \delta^*(p, (x_1 \cdots x_{n_1})x_n) &= \delta(\delta^*(p, x_1 \cdots x_{n_1}), x_n) \end{aligned}$$

donde nil es la palabra vacía.

Notemos que la definición dice que el autómata finito es *determinista*. Esto quiere decir que la función de transición está bien definida, por lo que no existen ambigüedades en cuanto a las decisiones que se deben de tomar al aplicar la función de transición. Un autómata finito lo podemos visualizar de varias maneras y se suele representar por medio de un diagrama de transiciones.

Un diagrama de transiciones es una colección finita de círculos, los cuales se pueden rotular para fines de referencia, conectados por flechas que reciben el nombre de arcos. Cada uno de dichos arcos también puede ser rotulado

y éstos pueden llevar una cadena de entrada. A uno de estos círculos se le asigna un apuntador el cual indica que dicho estado es el estado inicial $i \in S$. De alguna forma debemos indicar los estados de aceptación (finales). Para esto es muy común utilizar un círculo extra dentro del círculo. Decimos que una cadena es aceptada si al analizar carácter por carácter llegamos al círculo doble.

Por ejemplo, para pasar del estado p al estado q cuando leamos una x de nuestro alfabeto Σ, dibujaríamos dos círculos unidos por un arco rotulado con la letra x (ver figura 1.1).

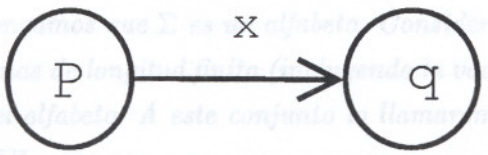

Figura 1.1: Diagrama de transiciones

Otra manera de visualizar los autómatas finitos es con tablas de transiciones. Una tabla de transiciones es una matriz bidimensional tal que sus elementos nos proporcionan información resumida de un diagrama de transiciones (ver figura 1.2).

Para elaborar una tabla de transiciones primero construimos una matriz tal que una fila de la matriz corresponda a los estados de la máquina y una columna de la matriz corresponda a los símbolos de nuestro alfabeto. El elemento que se encuentra en la fila m y en la columna n es el estado que se alcanzaría en el diagrama al dejar el estado m a través de leer un arco rotulado con el símbolo n; es decir, si leemos una n de la cadena de entrada cambiamos de estado.

	letra	digito	FDC
1	3	2	error
2	error	error	error
3	3	3	ok

Figura 1.2: Tabla de transiciones.

1.2. Lenguajes que aceptan los autómatas finitos

Definición 1.3. *Definimos la longitud de una cadena w como el número de símbolos que contiene, y la representaremos por $|w|$.*

Por ejemplo, para la cadena w=xyz, $|w|$=$|xyz|$=3.

Definición 1.4. *Definimos la cadena vacía como aquella cadena que no contiene ningún carácter y la denotaremos por λ.*

Definición 1.5. *Supongamos que Σ es un alfabeto. Consideraremos al conjunto de todas las cadenas de longitud finita (incluyendo la vacía) que pueden construirse a partir del alfabeto. A este conjunto lo llamaremos diccionario y lo denotaremos por Σ^*.*

Por ejemplo si Σ=$\{a, b\}$, entonces Σ^*=$\{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$. A un subconjunto $A \subseteq \Sigma^*$ lo llamaremos lenguaje (del alfabeto Σ). Si M es una autómata finito determinista, la colección de todas las cadenas que acepta M es un lenguaje, al que denotaremos por $L(M)$. Un lenguaje que es aceptado por un autómata finito es llamado *lenguaje regular.*

Ejemplo 1.3. Sea $\Sigma = \{a, b, c, \ldots, z\}$. Entonces

$$\Sigma^* = \{ \text{ cadenas minúsculas de longitud finita } \}$$

Un ejemplo de un lenguaje sobre Σ es

$$L = \{w \in \Sigma^* | w \text{ está en el diccionario de mi casa}\}.$$

Este lenguaje evidentemente es finito, y es un subconjunto propio del lenguaje español y además es único.

Ejemplo 1.4. Sea $\Sigma = \{a, b, \ldots; z, A, B, \ldots, Z, 0, 1, \ldots, 9, spc\}$ y sea

$$L = \{w \in \Sigma^* | w \text{ empieza con } a \text{ o con } A \text{ o } \ldots z \text{ o } Z \text{ y } |w| \leq 500 \times 2^{10}\}.$$

L es el conjunto de las palabras aceptadas por el lenguaje *Maple.*

A la colección que no tiene cadenas le llamaremos *lenguaje vacío*, y lo representamos por $\emptyset$.

Ejemplo 1.5. Veamos algunos ejemplos de lenguajes regulares sobre $\Sigma = \{a, b, c\}$
$L_1 = \{a, ab, abc\} \cup \{a\} \cup \{a\}\{b\} \cup \{a\}\{b\}\{c\}$;
$L_2 = \{w \in \Sigma^* |\ longitud(w)$ es un múltiplo de $3\} = (\Sigma\Sigma\Sigma)^*$

Teorema 1.1. *Dado un alfabeto Σ, existe un lenguaje que no es aceptado por M, donde M es cualquier autómata finito determinista.*

Demostración. Supongamos primero que los arcos de nuestros autómatas finitos están siempre rotulados con símbolos de nuestro alfabeto Σ, podemos suponerlo, pues si hubiera símbolos fuera de nuestro alfabeto no tendría efecto alguno.
Por otro lado, la colección de autómatas finitos es numerable, pues podemos hacer una lista en forma sistemática de los autómatas finitos con n estados, n=1, 2, ..., es decir, los autómatas con un estado, dos estados, etc.
Por otro lado el conjunto de lenguajes de Σ es no numerable ya que el conjunto Σ^* tiene una cantidad no numerable de subconjuntos. Por lo tanto, existen más lenguajes que autómatas finitos deterministas. Por lo tanto, como cada autómata finito determinista acepta un solo lenguaje, deben existir lenguajes que no sean aceptados por autómatas finitos deterministas. □

El teorema anterior nos asegura la existencia de lenguajes que no son regulares. Para poder dar un ejemplo de un lenguaje que no sea regular veamos la siguiente definición.

Definición 1.6. *Si w es una cadena, definimos w^n (donde n es un entero no negativo) como $w^n = ww\cdots w$.*

Teorema 1.2. *Si un lenguaje regular contiene cadenas de la forma $x^n y^n$ para enteros n arbitrariamente grandes, entonces debe contener cadenas de la forma $x^m y^n$, donde m y n no son iguales.*

Demostración. Supongamos que M es un autómata finito determinista tal que $L(M)$ contiene cadenas de la forma $x^n y^n$ para n arbitrariamente grande. Entonces podemos encontrar un entero k mayor que el número de estados de M y tal que $x^k y^k$ se encuentre en $L(M)$, pues k puede ser tan grande como queramos. Puesto que hay más símbolos en x^k que estados de M entonces al llevar a cabo el proceso de leer la cadena $x^k y^k$ tendremos que alguno de los estados de M se va a recorrer más de una vez antes de llegar a alguna de las y. Durante la lectura, alguna de las x recorrerá una ruta circular del diagrama de transiciones. Sea j el número de x leídas al recorrer esta ruta, entonces la máquina puede aceptar la cadena $x^{k+j} y^k$. Por lo tanto existe un entero positivo $m = k + j$ que no es igual a k, tal que $x^m y^k$ se encuentra en $L(M)$. □

Ejemplo 1.6. Veamos algunos ejemplos de lenguajes que no son regulares. Sobre el alfabeto $\Sigma = \{a, b\}$

- $L_1 = \{a^n b^n | n \in \mathbb{N}\}$
- $L_2 = \{ww | w \in \Sigma^*\}$

Sobre el alfabeto $\Sigma = \{a\}$

- $L_3 = \{a^k | k \text{ es primo}\}$

Sobre el alfabeto $\{(,), [,]\}$

- $L_4 = \{$ Las cadenas de paréntesis bien balanceadas $\}$

Existe otra clase de máquinas un poco más flexibles que los autómatas finitos deterministas. Este tipo de máquinas se llaman *autómatas finitos no deterministas* y, como veremos a continuación, su definición es un poco diferente, más sin embargo probaremos más adelante que los *AFD* y los autómatas no deterministas son equivalentes. Esto quiere decir que el poder de los autómatas es limitado y no varía si introducimos un no determinismo. Es decir, los autómatas no deterministas sí aceptan ambigüedades a la hora de decidir cuando se trata de pasar de un estado a otro.

Definición 1.7. *Un autómata finito no determinista* (AFND) *consiste en una quíntupla* (S, Σ, ρ, i, F) *donde:*

1. S *es un conjunto finito de estados*
2. Σ *es el alfabeto de la máquina*
3. ρ *es un subconjunto de* $S \times \Sigma \times S$
4. i *(un elemento de* S*) es el estado inicial*
5. F *(un subconjunto de* S*) es el conjunto de estados de aceptación*

Teorema 1.3. *Para cada autómata finito no determinista, existe un autómata finito determinista que acepta exactamente el mismo lenguaje.*

Demostración. Supongamos que M es el autómata finito no determinista definido por la quíntupla (S, Σ, ρ, i, F). El objetivo es demostrar la existencia de un AFD que acepta exactamente las mismas cadenas que M. Definamos otro autómata, M', con la quíntupla $(S', \Sigma, \delta, i', F')$, donde $S' = P(S)$, $i' = \{i\}$, F' es el conjunto de subconjuntos de S que contienen por lo menos un estado de F, y δ es la función de $S \times \Sigma$ a S' tal que para cada x en Σ y $s' \in S'$, $\delta(s', x)$ es el conjunto de todo S en S tal que (u, x, s) está en ρ para alguna u en s' (es decir, $\delta(S', x)$ es el conjunto de todos los estados de S a los que es posible llegar desde un estado en s' siguiendo un arco con etiqueta x). Podemos observar que como δ es una función, M' es un autómata finito determinista .

Lo que falta es mostrar que M y M' aceptan exactamente las mismas cadenas. Para esto, procedamos por inducción para mostrar que para cada $n \in \mathbb{N}$ se cumple la observación formulada al final del teorema.

□

Observación 1.1. Para cada ruta en M del estado i al estado S_n, que recorre arcos rotulados $w_1, w_2, \ldots, w_n$, existe una ruta en M' del estado i' al estado S'_n que recorre los arcos rotulados $w_1, \ldots, w_n$, de modo que $S_n \in S'_n$; y a la inversa, para cada ruta en M' de i' a S'_n recorriendo arcos rotulados $w_1, \ldots, w_n$ y cada $S_n \in S'_n$ existe una ruta en M de i a S_n que recorre arcos rotulados $w_1, \ldots, w_n$.

A partir de esto, se debe deducir que para cualquier ruta en M que vaya de i a un estado de aceptación y recorra arcos rotulados $w_1, \ldots, w_n$, debe existir una ruta en M' de i' a un estado de aceptación, que siga los arcos rotulados $w_1, \ldots, w_n$ y viceversa. Por lo tanto, M y M' deben aceptar el mismo lenguaje.

Demostración. Consideremos el caso $n = 0$. Aquí S_n debe ser i y S'_n debe ser $i' = \{i\}$, por lo tanto se tiene el resultado.

Supongamos la observación verdadera para un $n \in \mathbb{N}$ y consideremos una ruta en M, de i a algún s_{n+1}, que recorre los arcos rotulados $w_1, w_2, \ldots, w_{n+1}$. Sea s_n el penúltimo estado de M por esta ruta. Así, $(s_n, w_{n+1}, s_{n+1}) \in \rho$. Por la hipótesis de inducción, existe una ruta en M', de i' a algún $s'_{n'}$ que recorre arcos rotulados $w_1, w_2, \ldots, w_{n+1}$ de modo que $s_n \in s'_n$. Puesto que $(s_n, w_{n+1}, s_{n+1}) \in \rho$, existe un arco en M' rotulado w_{n+1} a un estado que contiene a s_{n+1}. Sea s'_{n+1} tal estado. Entonces existe una ruta en M', de i' a $s'_{n+1'}$, que recorre los arcos rotulados $w_1, w_2, \ldots, w_{n+1}$, de modo que $s_{n+1} \in s'_{n+1}$.

A la inversa, considere una ruta en M', de i' a algún $s'_{n+1'}$, que recorre los arcos rotulados $w_1, w_2, \ldots, w_{n+1}$, y sea s'_n el penúltimo estado de esta ruta. Luego, por inducción, para cada $s_n \in s'_n$ debe existir una ruta en M, de i a s_n que recorre los arcos rotulados $w_1, w_2, \ldots, w_n$. Pero $\delta(s'_{n'}, w_{n+1}) = s'_{n+1'} \in \rho$. Por lo tanto, para cada s en $s'_{n+1'}$, existe una ruta en M, de i a s, que recorre arcos rotulados $w_1, w_2, \ldots, w_{n+1}$, como se requiere. □

1.3. Gramáticas regulares

En este capítulo estudiaremos las gramáticas regulares. Veamos el siguiente ejemplo:

$$S \to xX$$
$$X \to yY$$
$$Y \to xX$$
$$Y \to \lambda$$

Veamos qué significa este diagrama. Este diagrama consta de símbolos, algunas letras mayúsculas, minúsculas, flechas y la cadena vacía λ. Las flechas corresponden a reglas de asignación, la letra S es un símbolo inicial (más adelante veremos a detalle y formalmente las definiciones). Es decir, comenzamos sustituyendo a S por su correspondiente equivalente.
Es decir, S cambia por xX. Ahora si aplicamos la regla de asignación que le corresponde a X, entonces tendríamos xyY y por último si cambiamos a Y por λ el resultado sería xy, y ya no tenemos más reglas para cambiar a xy. Es decir, llevando a cabo una sucesión de aplicaciones de las reglas de producción, hemos generado la cadena xy. Cabe notar que la cadena fue generada utilizando unos pasos muy particulares. Sin embargo, podríamos generar la cadena $xyxy$ aplicando las reglas en otro orden, pero siempre aplicando primero la regla que cambia S por su correspondiente equivalente.

A este conjunto de letras y reglas se le llama *gramática.* A las letras mayúsculas se les llama símbolos no terminales, a las letras minúsculas se les llama símbolos terminales, a la letra S se le llama símbolo inicial y a las sustituciones se les llama reglas de producción.

Definición 1.8. *Una gramática se define como una cuádrupla (V, T, S, R), donde V es un conjunto finito de no terminales, T es un conjunto finito de terminales, S es el símbolo inicial y R es un conjunto finito de reglas de escritura o reglas de producción.*

En general, los lados derechos e izquierdos de las reglas de escritura pueden constar de cualquier combinación de terminales y no terminales, sólo con una condición: siempre el lado izquierdo debe contener al menos un no terminal.

Algunas veces denotaremos a los símbolos no terminales entre corchetes y a los terminales solos. Se dice que una gramática genera una cadena si, al comenzar con el símbolo de inicio, se puede producir esa cadena aplicando sucesivamente las reglas de producción del conjunto R. A esta secuencia de pasos le llamaremos *derivación.*

Definición 1.9. *Una gramática regular es una gramática cuyas reglas de escritura se apegan a las siguientes restricciones:*

1. *El lado izquierdo de cualquier regla de escritura consiste de un único no terminal*
2. *El lado derecho de cualquier regla de escritura debe ser un terminal seguido por un no terminal, o un solo terminal, o la cadena vacía*

La importancia de las gramáticas regulares está en que los lenguajes generados por estas gramáticas son aquellos que aceptan los autómatas finitos.

Hasta ahora hemos visto que los lenguajes regulares son los lenguajes que son aceptados por autómatas finitos y los generados por las gramáticas regulares. En esta sección veremos que existe otra forma de visualizar a los lenguajes regulares.

Definición 1.10. *Sea Σ un alfabeto, si L_1 y L_2 son dos lenguajes regulares sobre Σ, entonces*

$$L_1 \cup L_2 = \{x \in \Sigma^* | x \in L_1 \ o \ x \in L_2\}$$

y el cual llamaremos el lenguaje regular unión.

Definición 1.11. *Sea Σ un alfabeto, si L_1 y L_2 son dos lenguajes regulares sobre Σ, entonces*

$$L_1 \cap L_2 = \{x \in \Sigma^* | x \in L_1 \ y \ x \in L_2\}$$

y el cual llamaremos el lenguaje regular intersección.

Definición 1.12. *Sea Σ un alfabeto, si L_1 y L_2 son dos lenguajes regulares sobre Σ L_1 y L_2, entonces*

$$L_1L_2 = \{wz \mid w \in L_1 \text{ y } z \in L_2\}$$

el cual llamaremos lenguaje regular concatenación.

Ejemplo 1.7. Sea $\Sigma = \{a, b\}$, sea $L_1 = \{a, ab\}$ y $L_2 = \{\lambda, b, bb\}$ entonces

$$L_1 \cup L_2 = \{a, ab, \lambda, b, bb\}$$

$$L_1 \cap L_2 = \emptyset$$

$$L_1L_2 = \{a, ab, b, bb, abb\}$$

Existen dos lenguajes regulares que en alguna ocasión se usarán: el lenguaje que contiene solamente la cadena vacía L=$\{\lambda\}$ y el lenguaje vacío L=$\emptyset$.
Sea Σ un alfabeto, L un subconjunto de Σ^*, definamos L^0=λ y $L^i = L^{i-1}L^i$ para $i \geq 0$. La *cerradura de Kleene* de L denotada por L^*, es el conjunto

$$L^* = \bigcup_{i=1}^{n} L^i$$

Las propiedades de los conjuntos también se cumplen para los lenguajes. Por ejemplo

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

para cualesquiera dos lenguajes L_1 y L_2. Existen también identidades para la concatenación y cerradura de Kleene.

Ejemplo 1.8. Si L_1 y L_2 son dos lenguajes sobre un alfabeto Σ, entonces

$$(L_1 \cup L_2)^* = (L_1^* \cup L_2^*)^* = (L_1^*L_2^*)^* \tag{1.1}$$

Demostración. Observemos que para cualesquiera dos lenguajes L_1 y L_2 se tiene que si

$$L_1 \subseteq L_2 \Rightarrow L_1^* \subseteq L_2^*$$

Para cualquier palabra en L_1^* la palabra es de la forma $w = w_1 w_2 \dots w_n$ donde las palabras w_k pertenecen a L_1, pero las w_k pertenecen también a L_2, por lo tanto $w \in L_2^*$.
Claramente $L_1 \subseteq L_1^*$ y $L_2 \subseteq L_2^*$, por lo tanto

$$(L_1 \cup L_2)^* \subseteq (L_1^* \cup L_2^*)^* \tag{1.2}$$

Ahora, se tiene que $\lambda \in L_2^*$, por lo tanto $L_1^* \subseteq L_1^* L_2^*$. Similarmente se tiene que $L_2^* \subseteq L_1^* L_2^*$, entonces tenemos que $L_1^* \cup L_2^* \subseteq L_1^* L_2^*$ por lo tanto tenemos que

$$(L_1^* \cup L_2^*)^* \subseteq (L_1^* L_2^*)^* \tag{1.3}$$

Finalmente, dado que cualquier palabra en $L_1^* L_2^*$ es una concatenación de palabras de L_1 seguidas de L_2 entonces podemos decir que es una palabra que está en $L_1 \cup L_2$. Cualquier palabra en $(L_1^* L_2^*)^*$ es otra vez una concatenación de dichas palabras y entonces se tiene

$$(L_1^* L_2^*)^* \subseteq (L_1 \cup L_2)^* \tag{1.4}$$

Ahora de las ecuaciones 1.2, 1.3, 1.4 se tiene el resultado. □

Definición 1.13. *Una expresión regular para un alfabeto Σ se define como sigue:*

- *El vacío $\emptyset$ es una expresión regular*
- *Cada miembro de Σ es una expresión regular*
- *Si p y q son expresiones regulares, entonces también lo es $(p \cup q)$*
- *Si p y q son expresiones regulares, entonces también lo es $(p \cdot q)$*
- *Si p es una expresión regular, entonces también lo es p^**

Figura 1.3: Cero operadores.

Teorema 1.4. *Dado un alfabeto Σ, los lenguajes regulares de Σ son exactamente los lenguajes representados por las expresiones regulares de Σ.*

Demostración. La demostración de este teorema se puede ver en [2] □

De ahora en adelante denotaremos al conjunto de las expresiones regulares por *ER*. Dentro de los autómatas finitos es posible extender a los autómatas finitos no deterministas a unos *AFND* con transiciones λ. Esto formalmente quiere decir que el autómata finito que se tenía, queda igual con excepción de δ, la función de transición, que ahora transforma $Q \times (\Sigma \cup \{\lambda\})$ a 2^Q. La intención es que $\delta(q, a)$ consista de todos los estados p tales que exista una transición con etiqueta a que vaya de q a p, siendo a el símbolo λ o un símbolo de Σ.

1.4. Teorema de Kleene

Teorema 1.5. *Sea r una expresión regular. Entonces existe un* AFND *con transiciones λ que acepta a $L(r)$.*

Demostración. Mostremos por inducción sobre el número de operadores de la expresión regular, r, que existe un *AFD*, M que posee un estado final y ninguna transición que parta de este estado final, tal que $L(M) = L(r)$.
Base: (Cero operadores) La expresión r debe ser λ, $\emptyset$ ó a para alguna a de Σ. El *AFND* que se muestra en la figura 1.3 satisface claramente las condiciones.

Inducción. (Uno o más operadores) Supongamos que el teorema es verdadero para expresiones regulares con menos de i operadores, $i \geq 1$. Sea r una expresión con i operadores. Existen tres casos, dependiendo de la forma de r.
CASO 1: $r = r_1 + r_2$. Tanto r_1 como r_2 deben de tener menos de i operadores. Por lo tanto, existen dos *AFND* $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$ y $M_2 =$

$(Q_2, \Sigma_2, \delta_2, q_2, f_2)$ con $L(M_1) = L(r_1)$ y $L(M_2) = L(r_2)$. Puesto que podemos renombrar estados de un *AFND* como queramos, podemos suponer que $Q_1 \cap Q_2$. Sea q_0 un nuevo estado inicial y f_0 un nuevo estado final. Constrúyase

$$M = (Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta_1, q_0, f_0),$$

en donde δ está definida por

1. $\delta(q_0, \lambda) = \{q_1, q_2\}$,
2. $\delta(q, a) = \delta_1(q, a)$ para $q \in Q_1 \setminus \{f_1\}$ y $a \in \Sigma_1 \cup \lambda$,
3. $\delta(q, a) = \delta_2(q, a)$ para $q \in Q_2 \setminus \{f_2\}$ y $a \in \Sigma_2 \cup \lambda$,
4. $\delta(f_1, \lambda) = \delta_1(f_2, \lambda) = f_0$

Notemos que por hipótesis inductiva no existen transiciones que partan de f_1 ó f_2 en M_1 ó M_2. Así pues, todos los movimientos de M_1 y M_2 están presentes en M.
En la figura 1.4 se describe la construcción de M. Cualquier trayectoria en el diagrama de transiciones de q_0 a f_0 debe empezar dirigiéndose a q_1 o a q_2 sobre λ. Si la trayectoria se dirige a q_1, puede seguir cualquier trayectoria en M_1 hacia f_1 y después ir a f_0 sobre λ. De manera similar, las trayectorias que comiencen dirigiéndose hacia q_2 pueden seguir cualquier trayectoria en M_2 hacia f_2, y de ahí dirigirse hacia f_0 sobre λ. Estas son las únicas trayectorias de q_0 a f_0. Se concluye de manera inmediata que existe una trayectoria con etiqueta x en M que va de q_0 a f_0 si y sólo si existe una trayectoria con etiqueta x en M_1 de q_1 a f_1, o una trayectoria en M_2 de q_2 a f_2. De aquí que $L(M) = L(M_1) \cup L(M_2)$ como se deseaba.
CASO 2: $r = r_1 r_2$. Sean M_1 y M_2 como en el Caso 1 y constrúyase

$$M = (Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, q_1, f_2),$$

en donde δ está definida como

1. $\delta(q, a) = \delta_1(q, a)$ para q en $Q_1 \setminus f_1$ y a en $\Sigma_1 \cup \lambda$

2. $\delta(f_1, \lambda) = q_2$

3. $\delta(q, a) = \delta_2(q, a)$ para q en Q_2 y a en $\Sigma_2 \cup \lambda$

Cada trayectoria en M de q_1 a q_2 es una trayectoria etiquetada con alguna cadena x y va de q_1 a f_1, seguida por el borde de f_1 a q_2 etiquetado con λ, seguido, a su vez, por una trayectoria etiquetada con alguna cadena y que va de q_2 a f_2. Por tanto, $L(M) = \{xy$ tal que x está en $L(M_1)$ y y está en $L(M_2)\}$ y $L(M) = L(M_1)L(M_2)$ como se quería demostrar.

CASO 3: $r = r^*$. Sea $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_2)$, y $L(M_1) = r_1$. Constrúyase

$$M = (Q_1 \cup \{q_0, f_0\}, \Sigma_1, \delta, q_0, f_0),$$

en el que δ está dada por

1. $\delta(q_0, \lambda) = \delta(f_1, \lambda) = \{q_1, f_0\}$

2. $\delta(q, a) = \delta_1(q, a)$ para q en $Q_1 \setminus f_1$ y a en $\Sigma_1 \cup \lambda$

La construcción de M se presenta en la figura 1.4. Cualquier trayectoria de q_0 a f_0 consiste en una trayectoria de q_0 a f_0 sobre λ o una trayectoria de q_0 a q_1 sobre λ, seguida de cierto número (posiblemente cero) de trayectorias de q_1 a f_1. Después se retrocede a q_1 sobre λ, cada trayectoria etiquetada con una cadena de $L(M_1)$, seguida, a su vez, por una trayectoria de q_1 a f_1 sobre una cadena de $L(M_1)$, después a f_0 sobre λ. Así pues, existe una trayectoria en M de q_0 a f_0 con etiqueta x si y sólo si podemos escribir $x = x_1 x_2 \ldots x_j$ para alguna $j \geq 0$ (el caso $j = 0$ significa $x = \lambda$) tal que cada x_1 esté en $L(M_1)$. De aquí que $L(M) = L(M_1)^*$ como se deseaba. □

Ejemplo 1.9. Construyamos un *AFND* para la expresión regular $\mathbf{01^* + 1}$.

Solución. Tomando en cuenta las reglas que se acaban de discutir, esta expresión regular se puede ver de la siguiente forma $\mathbf{(0(1^*)) + 1}$, así que es de la forma $r_1 + r_2$, en donde $r_1 = \mathbf{01^*}$ y $r_2 = \mathbf{1}$. El autómata correspondiente a r_2 es muy simple; es de la forma como el que se presenta en la figura 1.5.

Podemos expresar r_1 como $r_3 r_4$, en donde $r_3 = \mathbf{0}$ y $r_4 = \mathbf{1^*}$. El autómata correspondiente a r_3 es como el de la figura 1.6.

A su vez, r_4 es r_5^*, en donde r_5 es $\mathbf{1}$ y su autómata es evidente.

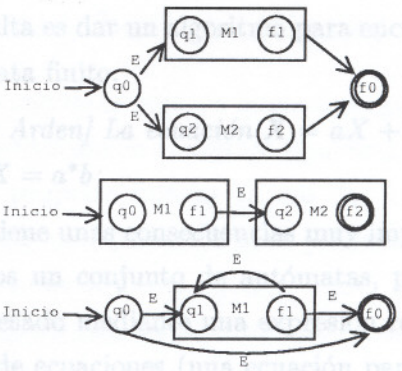

Figura 1.4: Diagrama del autómata M.

Figura 1.5: Autómata de **1**

1.5. Puntos fijos de ecuaciones lineales

Sean dos expresiones regulares A, B, sea $P = \{X \in ER | X = AX + B\}$ expresiones regulares que son puntos fijos de la función $XAX + B$

Observación 1.2. Las siguientes relaciones son verdaderas:

1. La expresión A^*B está en P
2. A^*B es una cota inferior de P

Con el teorema de Kleene antes probado tenemos una herramienta muy potente para poder construir autómatas finitos dada una expresión regular.

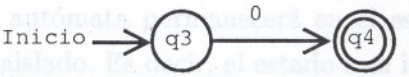

Figura 1.6: Autómata de **0**

Ahora lo único que nos falta es dar un algoritmo para encontrar expresiones regulares dado un autómata finito.

Teorema 1.6. *[Regla de Arden] La ecuación $X = aX + b$ con $a \neq \lambda$ tiene una única solución y es $X = a^*b$*

Este último teorema tiene unas consecuencias muy importantes, pues en el momento que tengamos un conjunto de autómatas, podremos saber el lenguaje que acepta expresado mediante una expresión regular, pues basta con plantear un sistema de ecuaciones (una ecuación para cada estado del autómata) y hacer una substitución de abajo hacia arriba para encontrar la expresión regular asociada al autómata.

1.6. Parte conexa de un autómata finito

Supongamos que tenemos un autómata finito M definido de la siguiente manera:

$$M = (\Sigma = \{0,1\}, S = \{a,b,c\}, \delta, i = \{a\}, F = \{b\})$$

donde δ está definida de la siguiente manera:

- $\delta(a,0) = a$
- $\delta(a,1) = b$
- $\delta(b,0) = b$
- $\delta(b,1) = b$
- $\delta(c,0) = c$
- $\delta(c,0) = a$

Una vez que el estado actual del autómata es b, para todo $i \in \Sigma$, $\delta(b,i) = b$, indica que el autómata permanecerá en el estado b y el estado c está completamente aislado. Es decir, el estado c es inaccesible.

Definición 1.14. *Sea $M = (S, \Sigma, \delta, i, F)$ un autómata finito, decimos que $q \in S$ es accesible si existe una palabra $w \in \Sigma^*$ tal que $\delta^*(i,w) = q$*

1.7. Sustituciones y homomorfismos

La clase de conjuntos regulares posee la interesante propiedad de ser cerrada con respecto a la sustitución en el sentido siguiente. Para cada símbolo a del alfabeto de algún conjunto regular R, sea R_a un conjunto regular particular. Supongamos que reemplazamos cada palabra $a_1, a_2, \ldots, a_n$ de R por el conjunto de palabras de la forma $w_1, w_2, \ldots, w_n$, en donde w_i es una palabra cualquiera de R_{ai}. El resultado, entonces, es siempre un conjunto regular. De manera más formal, una sustitución f es una transformación de un alfabeto Σ en subconjuntos de Δ^*, para algún alfabeto Δ. Por tanto f asocia un lenguaje con cada símbolo de Σ. La transformación f se extiende a cadenas de la manera siguiente:

1. $f(\lambda) = \lambda$
2. $f(xa) = f(x)f(a)$

La transformación f se extiende a lenguajes mediante la definición

$$f(L) = \bigcup_{x \in L} f(x)$$

Ejemplo 1.10. Sean $f(0) = a$ y $f(1) = b^*$. Esto es, $f(0)$ es el lenguaje $\{a\}$ y $f(1)$ es el lenguaje que consiste en todas las cadenas de bs. Entonces $f(010)$ es el conjunto regular ab^*a. Si L es el lenguaje $0^*(0+1)1^*$, entonces $f(L)$ es $a^*(a+b)^*(b^*)^* = a^*b^*$.

Teorema 1.7. *La clase de conjuntos regulares es cerrada bajo la sustitución*

Demostración. Sea $R \subset \Sigma^*$ un conjunto regular y, para cada $a \in \Sigma$, sea $R_a \subseteq \Delta^*$ también un conjunto regular; $f : \Sigma \to \Delta^*$ es la sustitución definida por $f(a) = R_a$. Seleccionemos expresiones regulares que representen a R y a cada R_a. Reemplacemos cada vez que se dé el símbolo a en la expresión regular para R por la expresión regular para R_a. Para demostrar que la expresión regular que resulta representa a $f(R)$, observemos que la sustitución de una unión, producto o cerradura es la unión, producto o cerradura

de la sustitución. (Por tanto, por ejemplo, $f(L_1 \cup L_2) = f(L_1) \cup f(L_2)$). Una simple inducción sobre el número de operadores en la expresión regular completa la prueba.

Notemos que en el ejemplo anterior calculamos $f(L)$ tomando la expresión regular de L como $0^*(1+0)1^*$ y sustituyendo a por 0 y b por 1. El hecho de que la expresión regular resultante sea equivalente a la expresión regular más simple a^*b^* es mera coindidencia.

Un tipo de sustitución que es de especial interés es el homomorfismo. Un *homomorfismo* h es una sustitución tal que $h(a)$ contiene una sola cadena para cada a. Por lo general tomamos $h(a)$ como la cadena misma, más que el conjunto que contiene a tal cadena. Es de utilidad definir la *imagen homomórfica inversa* de un lenguaje L como

$$h^{-1}(L) = \{x | h(x) \in L\}$$

También utilizamos, para la cadena w:

$$h^{-1}(w) = \{x | h(x) = w\}$$

□

Teorema 1.8. *La clase de conjuntos regulares es cerrada con respecto a los homomorfismos y los homomorfismos inversos.*

Demostración. La cerradura con respecto a los homomorfismos se sigue inmediatamente a partir de la cerradura con respecto a la sustitución, ya que cada homomorfismo es una sustitución, en la que $h(a)$ tiene sólo un elemento.

Para mostrar la cerradura con respecto a los homomorfismos inversos, sean $M = (Q, \Sigma, \delta, q_0, F)$ un *AFD* que acepte a L, y h el homomorfismo de Δ a Σ^*. Construimos un *AFD* M' que acepta a $h^{-1}(L)$ mediante la lectura del símbolo a de Δ y que simule a M sobre $h(a)$. Formalmente, sea $M' = (Q, \Delta, \delta', q_0, F)$ y defínase $\delta'(q, a)$, para q en Q y a en Δ, como $\delta(q, h(a))$.

Notemos que $h(a)$ puede ser una cadena larga o ϵ, pero δ está definida, por extensión, sobre todas las cadenas. Es fácil mostrar, por inducción sobre $|x|$, que $\delta'(q_0, x) = \delta(q_0, h(x))$. Por consiguiente, M' acepta a x si y sólo si M acepta a $h(x)$. Esto es, $L(M') = h^{-1}(L(M))$. □

1.8. Cocientes de lenguajes

Fijemos ahora nuestra atención a la última propiedad de cerradura de los conjuntos regulares, que se demostrará en esta sección. Defínase el *cociente* de los lenguajes L_1 y L_2 que se escribe L_1/L_2, como

$$\{x| \text{ existe } y \in L_2 | xy \in L_1\}.$$

Teorema 1.9. *La clase de conjuntos regulares es cerrada con respecto al cociente con conjuntos arbitrarios.*

Demostración. Sea $M = (Q, \Sigma, \delta, q_0, F)$ un autómata finito que acepta algún conjunto regular R, y sea L un lenguaje cualquiera. El cociente R/L es aceptado por un autómata finito $M' = (Q, \Sigma, \delta, q_0, F')$ que se comporta como M excepto que los estados finales de M' son todos los estados q de M tales que existe y en L para el cual $\delta(q, y)$ está en F. Entonces $\delta(q_0, x)$ está en F' si y sólo si existe y tal que $\delta(q_0, xy)$ está en F. Por tanto M' acepta a R/l. □

Capítulo 2

Sistemas de planeación y esquemas de localización

2.1. Sistemas de planeación

Una formulación simple del problema de planeación contiene las siguientes tres entradas:

1. una descripción del *estado inicial* del conjunto en algún lenguaje formal,
2. una descripción de la meta a llegar (es decir, el comportamiento deseado) en algún lenguaje formal, y
3. una descripción de las posibles acciones que pueden ser ejecutadas. Esta última descripción es comúnmente llamada teoría de dominio

La salida es una sucesión de acciones las cuales cuando son ejecutadas en un conjunto (mundo) que satisface la descripción inicial, llegarán a la descripción meta. Veamos algunos ejemplos basados en la formalización anterior.

Ejemplo 2.1 (Juego de gato).

Según la formulación vista necesitamos establecer una descripción inicial del problema. En el caso de nuestro ejemplo tenemos que la descripción inicial del problema es un arreglo de tres por tres vacío como se muestra en la figura 2.1.

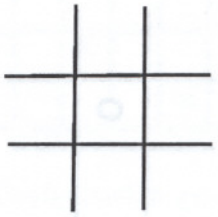

Figura 2.1: Tablero vacío

Siguiendo los pasos de la formulación necesitamos una meta que alcanzar. En este caso la meta del jugador es la de colocar tres símbolos iguales ya sea horizontal, vertical o diagonalmente para ganar. Se muestra un ejemplo en la figura 2.2.

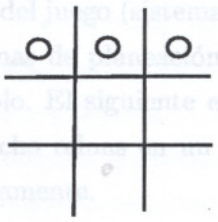

Figura 2.2: Un tablero ganador

Para continuar debemos proponer un conjunto de acciones posibles a ser ejecutadas. Como es sabido, en el juego de gato se requiere de dos jugadores, digamos A y B en donde los jugadores escogen una constante cada uno, digamos X y O respectivamente. Las acciones posibles son:

1. Colocar una X en una casilla del arreglo o

2. Colocar una O en una casilla del arreglo

Una vez que se tiene la descripción inicial definida, se procede a jugar. Se arranca deliberando si el jugador A o el B ejecutan una de las acciones permitidas. Una vez ejecutada la acción, digamos la 1, la descripción inicial se modifica y la nueva descripción es como la de la figura 2.3.

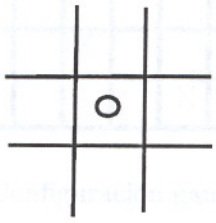

Figura 2.3: Primera etapa

Es necesario llevar un conteo de las acciones ejecutadas. Sucesivamente se aplican las acciones hasta que sucede una de dos situaciones: se alcanzó una descripción *meta* o no se alcanzó y el tablero puede estar lleno o no. Aplicando las acciones una a una, el tablero se va modificando y a su vez se va cambiando la descripción actual del juego (sistema) hasta terminar. Veamos en la siguiente sección los sistemas de planeación desde un punto de vista más formal. Veamos otro ejemplo. El siguiente ejemplo está basado en el problema conocido de colocar ocho reinas en un tablero de ajedrez de tal manera que no se ataquen mutuamente.

Ejemplo 2.2 (Problema de las ocho reinas).

El problema de las ocho reinas consiste en colocar ocho reinas en un tablero de ajedrez sin que se ataquen mutuamente. Necesitamos un tablero de ajedrez y ocho reinas. Notemos que el problema tiene al menos una solución (véase la figura 2.4).

Según la formalización antes vista necesitamos un estado inicial. De igual manera que en nuestro ejemplo anterior, nuestro estado inicial será un tablero de ajedrez. Para seguir adelante, necesitamos un conjunto de reglas

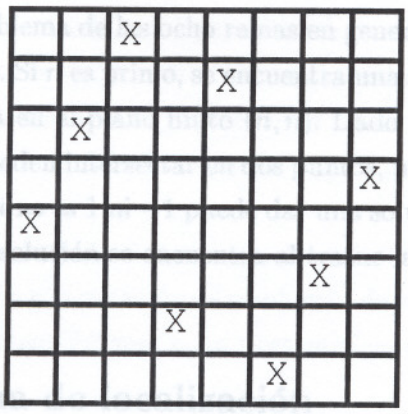

Figura 2.4: Configuración ganadora

o acciones válidas. Para esto tomemos como reglas las reglas del ajedrez, es decir, una reina ataca a otra reina si acaso están en la misma fila o en la misma columna o en la misma diagonal. Podemos formar una regla cuyo efecto sea el de colocar una reina en la posición (x, y) la cual llamaremos COLOCAREINA(X,Y). Por último necesitamos un conjunto de estados finales. En nuestro caso nuestro conjunto (digamos F) estará formado por todas las configuraciones ganadoras. Algo que debemos notar es que para aplicar las acciones el sistema de planeación se vale de algunos predicados que son de gran ayuda para dar información del sistema.

Por ejemplo, si el sistema de planeación quisiera aplicar la regla COLOCAREINA(X,Y) entonces el sistema de planeación debería de ser capaz de reconocer si acaso la posición $(1, 1)$ está vacía y además no hay una reina atacando esa pieza. Para eso hacemos uso de algunos predicados que nos dan información, por ejemplo ESTAVACIO(X,Y) o ISCASILLAATACADA(X,Y). De esta manera tenemos resuelto el problema de aplicar una acción y la acción quedaría como una entidad atómica, es decir, no necesita de otras cosas para aplicarse. La acción sólo se aplica, pero la verificación no corre por cuenta de ella.

Podemos hablar del problema de las ocho reinas en general, consideremos el problema de las n reinas. Si n es primo, se encuentra una solución facilmente al trazar una línea recta en el plano finito (n, n). Dado que ningún par de líneas rectas dadas se pueden intersectar en dos puntos, una línea recta de la forma $y = ax + b$ donde a no es 1 ni -1 puede dar una solución, por ejemplo, para $n = 7$, $y = 2x$ y la solución se encuentra al trazar las dos líneas rectas antes mencionadas.

2.2. Un esquema de localización

El campo de la planeación por medio de la Inteligencia Artificial (AI Planning) busca construir algoritmos de control que permitan a un agente el sintetizar una ruta de acciones que alcancen cierta meta. Aun cuando los investigadores han estudiado la planeación desde los principios de la inteligencia artificial, recientes desarrollos han revolucionado el campo. Dos tratamientos han atraído en particular la atención:

- El algoritmo de planeación de dos fases llamado `Graphplan`, y
- Métodos de transformación de problemas de planeación a problemas proposicionales para después usar los algoritmos más conocidos de *SAT*

Estas dos propuestas tienen mucho en común y los dos han crecido mucho en desarrollo. El nivel actual de desempeño del planeador (performance) es bueno. Como un ejemplo muy en particular tenemos el planeador BLACKBOX (véase [4]) que requiere sólo de seis minutos para encontrar una ruta de 105 acciones en un conjunto de 10^6 posibles estados.

2.2.1. Algoritmo `graphplan`

El algoritmo `graphplan` (véase [4]) consiste de dos partes: La expansión de la gráfica y la extracción de la solución. La fase de expansión extiende una *gráfica de planeación* (planning graph) hacia adelante hasta encontrar una

condición necesaria (pero insuficiente) para la existencia del plan. La fase de extracción de la solución ejecuta una búsqueda hacia atrás en la gráfica en busca de un plan que resuelva el problema; si no se encuentra ninguna solución, entonces se repite el ciclo de expansión de la gráfica.

Comencemos considerando la formulación inicial del `Graphplan` y restrinjamos nuestra atención a la notación de resolución de problemas *STRIPS*. En otras palabras, las precondiciones y las postcondiciones de acciones son conjunciones de literales.

2.2.2. Expandiendo la gráfica

La gráfica de planeación contiene dos tipos de nodo, nodos de proposición y nodos de acción colocados en niveles. Los niveles pares contienen nodos de proposición, el nivel cero contiene las condiciones iniciales del problema y los niveles impares corresponden a las instancias de las acciones. Hay un solo nodo por cada acción cuyas precondiciones están presentes (y son mutuamente consistentes) en el nivel anterior. Los segmentos conectan los nodos de proposición con las acciones (en el siguiente nivel).

Notemos que la gráfica representa acciones *paralelas* en cada acción del nivel. Esto significa que una gráfica de planeación con k niveles de acción representa un plan con más de k acciones. Sin embargo, el tener dos acciones al mismo nivel no implica que es posible ejecutar ambas a la vez. Existe una relación de exclusión mutua entre nodos al mismo nivel que llamaremos *mutex*. Definimos esta relación recursivamente como sigue (véase también la figura 2.5):

- Dos acciones en el nivel i son *mutex* si
  - *Efectos inconsistentes:* el efecto de una acción es la negación del efecto de otra acción, o
  - *Interferencia:* una acción borra la precondición de otra, o

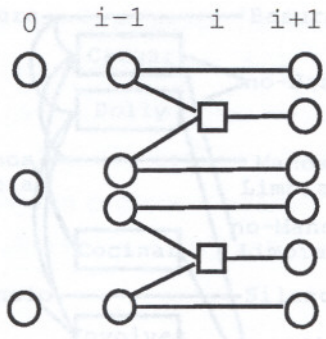

Figura 2.5: Nodos

- *Necesidades:* las acciones tienen precondiciones que son mutuamente exclusivas al nivel $i-1$

- Dos proposiciones al nivel i son mutex si una es la negación de otra, o todos los caminos de acceso a la proposición son mutex a pares (*Inconsistencia de soporte*)

Por ejemplo, consideremos el problema de preparar una cena para una persona (ver la figura 2.6). La meta es tirar la BASURA en su lugar, PREPARAR la cena y envolver un REGALO (ver la figura 2.6). Hay cuatro posibles acciones: COCINAR, ENVOLVER, CARGAR y ARRASTRAR. Cocinar requiere MANOSLIMPIAS y se obtiene CENA. ENVOLVER tiene como precondición SILENCIO (pues es un regalo sorpresa) y se obtiene un REGALO. CARGAR elimina la BASURA, pero puede producir mal olor lo cual niega { MANOSLIMPIAS }. La acción final, ARRASTRAR, también elimina la BASURA, pero produce ruido excesivo y niega SILENCIO; alguna otra proposición es falsa.

La figura muestra la gráfica de planeación para el problema de la cena expandido desde el nivel cero a través de una acción al nivel de proposición. Notemos que la acción CARGAR es *mutex* con la persistencia de BASURA porque tienen efectos inconsistentes. ARRASTRAR es inconsistente con ENVOLVER por interferencia, pues ARRASTRAR elimina SILENCIO. Al nivel

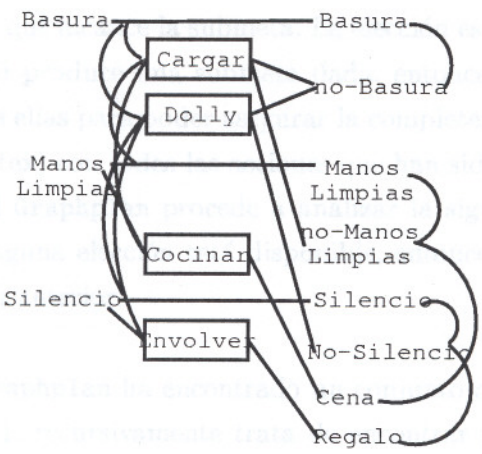

Figura 2.6: Problema de la cena

dos, no SILENCIO es *mutex* con REGALO por inconsistencia.

Como todas estas literales están presentes en el nivel dos, y no son *mutex* mutuamente, hay una esperanza de que el plan exista. En este caso, la segunda fase del `Graphplan` es ejecutado: Extracción de la solución.

2.2.3. Extracción de la solución

Supongamos que el `Graphplan` está tratando de generar un plan para una meta con n submetas, y (como en nuestro ejemplo) se ha expandido la gráfica de planeación a un nivel par, i, en donde todas las proposiciones meta están presentes y no son *mutex* a pares. Esto que acabamos de ver es una condición necesaria (pero insuficiente) para la existencia del plan. Por lo tanto, el `Graphplan` trata de extraer una solución y con un proceso de búsqueda hacia atrás en cadena (backward chaining search) busca si existe tal plan.

La extracción de la solución busca un plan considerando cada una de las n submetas en turno. Para cada literal al nivel i, `Graphplan` escoje una

acción a al nivel $i-1$ que alcance la submeta. La elección es un punto atrás: Si más de una acción produce una submeta dada, entonces el **Graphplan** debe considerar todas ellas para poder asegurar la completez. Si a es consistente (es decir, nomutex) con todas las acciones que han sido elegidas hasta ese nivel, entonces el **Graphplan** procede a analizar la siguiente submeta. De otra forma, si ninguna elección está disponible, entonces el **Graphplan** regresa a una elección anterior.

Después que el **Graphplan** ha encontrado un conjunto de acciones consistentes al nivel $i-1$, recursivamente trata de encontrar un plan para el conjunto formado por la unión de todas las precondiciones de las acciones al nivel $i-2$. Dado que estamos hablando de recursividad, debemos de analizar el caso base. El caso base para esta recursión es el nivel cero: si las proposiciones están presentes, entonces el **Graphplan** ha encontrado una solución. De otra forma, si el "backtracking" falla en todas las combinaciones posibles de acciones con submetas (en cada nivel), entonces el **Graphplan** extiende la gráfica de planeación con una acción adicional y niveles de proposición y trata de extraer otra vez una solución. Es decir, en esta última parte se especifica que el **Graphplan** de alguna manera trata de extraer una solución al problema, aun si esto implica tener que agregar más grados de libertad al problema.

En nuestro ejemplo, hay tres submetas al nivel dos. NO-BASURA se alcanza con CARGAR y por ARRASTRAR; CENA se alcanza por medio de COCINAR, y REGALO por medio de ENVOLVER. Así, el **Graphplan** debe considerar dos conjuntos de acciones al nivel uno:
{ CARGAR, COCINAR, ENVOLVER } y { ARRASTRAR, COCINAR, ENVOLVER }. Pero desafortunadamente ninguno de estos dos conjuntos es consistente, pues CARGAR es mutex con COCINAR mientras que ARRASTRAR es mutex con ENVOLVER. Así, la extracción de la solución falla, y el **Graphplan** extiende la gráfica de planeación al nivel cuatro como se muestra en la figura 2.7.

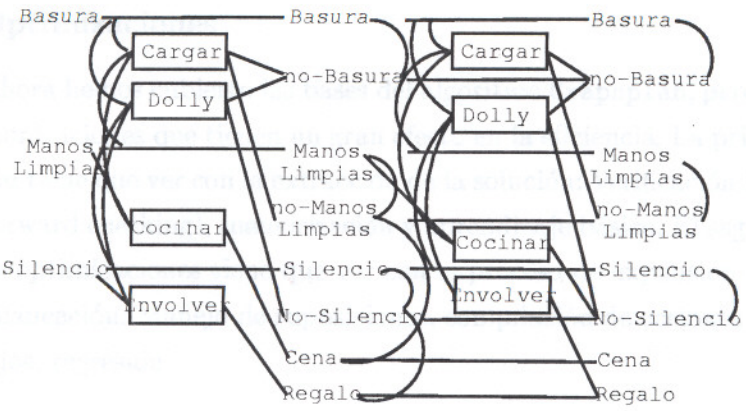

Figura 2.7: Gráfica extendida

Note la diferencia entre el nivel dos y el nivel cuatro de la gráfica de planeación. A pesar que no hay nuevas literales en el nivel cuatro, hay menos relaciones mutex. Por ejemplo, no hay mutex entre CENA y LIMPIAR MANOS en el nivel cuatro. La diferencia más importante está en el nivel tres donde hay cinco acciones adicionales codificando la posible persistencia de literales alanzadas en el nivel dos. Esto significa que cada submeta tiene un soporte adicional de acciones cuando se considere el proceso de extracción de la solución. Específicamente:

- ¬BASURA se soluciona con CARGAR, ARRASTRAR, y un acción
- CENA se alcanza con COCINAR y una acción
- REGALO se alcanza por medio de ENVOLVER y una acción

Entonces, la extracción de la solución necesita considerar $3 \times 2 \times 2 = 12$ combinaciones de acciones al nivel tres en lugar de $2 \times 1 \times 1 = 2$ combinaciones durante el anterior intento de extraer la solución. Así pues, esta flexibilidad que se agrega permite extraer la solución. Hay muchas combinaciones que funcionan.

2.2.4. Optimizaciones

Hasta ahora hemos cubierto las bases del algoritmo `Graphplan`, pero hay muchas optimizaciones que tienen un gran efecto en la eficiencia. La primera optimización tiene que ver con la extracción de la solución: Verificación hacia adelante (forward checking), memorización y aprendizaje básico. El segundo conjunto de optimizaciones tiene que ver con el proceso de expansión de la gráfica de planeación: Manejo de suposiciones, compilación de acciones para remover flujos, regresión.

El beneficio obtenido por cada una de estas optimizaciones depende específicamente del problema de planeación que estemos tratando. En el peor de los casos, el tiempo de expansión de la gráfica de planeación es *polinomial* mientras que el de la extracción de la solución es exponencial. Sin embargo, en muchos problemas de planeación es el tiempo de expansión el que domina; cada una de las optimizaciones comentadas es importante.

Extracción de la solución como satisfacción de restricciones
Observando la conexión que existe entre el algoritmo de extracción de solución del `Graphplan` y los problemas de satisfacción de restricciones, podemos transferir varios puntos de los problemas de satisfacción *CSP* al campo de la planeación. Hay varias posibles formulaciones, pero la más simple es en términos de un *CSP* dinámico; es decir, un problema de satisfacción en el cual el conjunto de variables y restricciones asociados cambian con base en la selección de los valores de las variables anteriores.
Existe un *CSP* variable para literales submetas en cada proposición para cada nivel después del nivel cero. El *Dominio* de una variable (es decir, su conjunto de posibles valores) es el conjunto de acciones válidas al nivel anterior. El conjunto de restricciones está definido por las relaciones mutex. Por ejemplo, consideremos el proceso de extracción de la solución para el nivel cuatro del ejemplo de la cena mostrado en la figura 2.7. Inicialmente, podemos crear un *CSP* variable para cada submeta al nivel cuatro: $V_{4,\neg Basura}$

toma un valor de {Cargar, Arrastrar, Mantener}, $V_{4,cena}$ toma un valor de {Cocinar, Mantener}, y $V_{4,Regalo}$ de {Envolver, Mantener}. Las asignaciones

$$V_{4,Basura} = Cargar$$

$$V_{4,Cena} = Mantener$$

$$V_{4,Regalo} = Envolver$$

corresponden a la primera parte de la solución mostrada en la figura 2.7. Una vez que la solución es encontrada para las variables en el nivel de proposición cuatro, las acciones correspondientes a los valores de las variables definen un problema *CSP* al nivel dos. Notemos que no hay requerimientos a efectuar nivel por nivel. En otras palabras, la descripción anterior de la extracción de la solución establece el encontrar un conjunto de acciones al nivel i antes de efectuar cualquier búsqueda al nivel $i-2$. Sin embargo, este orden no es necesario y es potencialmente ineficiente. Por ejemplo, el planeador *BLACKBOX* toma la gráfica de planeación, lo compila a una problema *SAT*, y usa veloces métodos estocásticos para ejecutar algo semejante al método de extracción de la solución en el cual hace saltos de nivel en nivel de una forma *avariciosa (greedy)*. Un algoritmo *avaricioso*, es un algoritmo que sigue un mismo patrón de ejecución desde el principio. La idea de los algoritmos *avariciosos* es seguir siempre un mismo procedimiento hasta que no se pueda aplicar más, y al final, verificar el resultado que produce. Tal vez no encuentre la solución o tal vez si, tal vez no sea el mejor método para solucionar un problema, pero muestra una manera de atacar un problema.

Por sí misma, la formulación *CSP* no es muy notoria, pero sugiere ciertas estrategias para hacer más veloz la búsqueda, como, *forward checking, dynamic variable ordering, memoization,* y *conflict-directed backjumping.*

- Cuando se asigna un valor a una variable, el CSP verifica que la elección sea consistente con todos lo valores previamente elegidos. Una mejor estrategia llamada *forward checking*, verifica además las variables que no han sido asignadas, reduciendo su dominio y eliminando

cualesquiera de los valores para los cuales se vuelve inconsistente. Si el dominio de cualquier variable no asignada se colapsa (es decir, se convierte en el conjunto vacío), entonces el programa de CSP debería efectuar un retroceso ("backtrack")

- El *ordenamiento dinámico de variables* se refiere a una clase de heurística de elección para saber qué valor se le va asignar a una variable. Por supuesto, eventualmente, todas las variables deben tener valores asignados, pero el orden en el cual son seleccionadas puede tener un gran impacto en la eficiencia. Notemos que si una variable sólo tiene una elección, entonces claramente lo mejor es asignarle ese valor inmediatamente. En general, una buena heurística es seleccionar la variable con menos valores restantes

- El artículo original del `Graphplan` [4] describe una técnica llamada *memoization* la cual almacena para posterior uso los resultados obtenidos de una búsqueda exhaustiva sobre conjuntos inconsistentes de submetas. Supongamos que la extracción de la solución es ejecutada en el nivel i e intentamos alcanzar submetas P, Q, R, y S en el nivel k (donde $k \leq i$). Si ninguna de las combinaciones de acciones para estas submetas nos produce consistencia, entonces el `Graphplan` almacena el conjunto $\{P, Q, R, S\}$ como un nivel $\neg$*bueno* para el nivel k. Después, si el `Graphplan` expande la gráfica de planeación al nivel $i + 2$ y una vez más buscamos la solución, probablemente intente alcanzar el mismo conjunto de cuatro submetas al nivel k pero esta vez ejecutará un "backtrack" inmediatamente en lugar de hacer la búsqueda exhaustiva. El proceso de *memoization* canjea espacio por tiempo, y aunque el espacio requerido puede ser grande, la mejora en la velocidad es significativa.

Como hemos dicho, el proceso de memoization es simplista. Existen procesos más sofisticados que han probado eficiencia en las resoluciones del SAT y el autor propone que se puede mejorar el proceso

de memoization. Con base al `Graphplan`, recientes trabajos por Kambhampati [5], [8] demuestran grandes avances en velocidad. La idea básica es determinar cual subconjunto de metas es responsable de la falla en un nivel dado y almacenarlo; si la extracción de la solución alguna vez regresa al nivel almacenado, entonces la falla es justificada. Este proceso nos lleva a un conjunto más chico de niveles $\neg buenos$; por ejemplo, podría ser el caso que las submetas P, Q, y S juntas sean inalcanzables a pesar de R

2.2.5. Suposición del mundo cerrado

La suposición del mundo cerrado establece que cualquier proposición que explícitamente no sea verdadera en el estado inicial, puede ser supuesta como falsa. Una forma sencilla de implementar esto en el `Graphplan` sería *cerrar* el nivel cero de la gráfica de planeación, es decir, agregar variables negativas para todas las posibles proposiciones que no fueron explícitamente establecidas como verdaderas.

Una mejor solución para manejar la suposición del mundo cerrado, es hacerlo de una manera *floja* (lazy), pues esto reduce el tamaño de la gráfica de planeación y reduce el costo de expansión. En este caso, el proceso consiste en crear el nivel cero de la gráfica de planeación agregando las proposiciones que son verdaderas en el estado inicial. Supongamos que una acción A requiere la precondición $\neg P$. Si $\neg P$ está en la gráfica de planeación al nivel $i-1$ se hace el trabajo usual. Sin embargo, si $\neg P$ no está desde el nivel $i-1$ debemos verificar si la negación (es decir, P) está presente al nivel cero. Si P está ausente del nivel cero, entonces agregar $\neg P$ al nivel cero y mantener las acciones mutex para acarrear $\neg P$ arriba al nivel actual. Con este simple acercamiento, no se necesitan cambios para la extracción de la solución.

Notemos que este tema de la suposición del mundo cerrado no contradice al ejemplo del problema de la cena porque ninguna de las acciones tienen precondiciones negativas. Y aunque se incluyó una literal negativa ($\neg Basura$) la proposición positiva estuvo presente en las condiciones iniciales. Así pues, debemos notar que es sumamente importante la suposición del mundo cerrado para muchos problemas de planeación.

2.3. Presentación de los sistemas de planeación

Formalmente, un *sistema de planeación* puede plantearse sobre una *signatura*, unión de los conjuntos de símbolos siguientes:

- *Ctes*: un conjunto de *constantes*, $Ctes = \{c_1, \ldots, c_k\}$,
- *Var*: un conjunto de *variables*, $Var = \{x_1, \ldots, x_v\}$,
- *Rnes*: un conjunto de *relaciones*, $Rnes = \{R_1, \ldots, R_r\}$. A cada símbolo de relación se le asocia una *aridad*, $ar(R) \in \mathbb{N}$,
- *Reglas*: un conjunto de *reglas lógicas*, $Reglas = \{\rho_1, \ldots, \rho_e\}$
- *Acc*: un conjunto de *acciones*, $Acc = \{A_1, \ldots, A_a\}$

Sobre la signatura definimos a los *términos* y a los *átomos*:

- Toda constante o variable es un *término*:

$$t \in Ctes \cup Var \;\Rightarrow\; t \in Térm$$

- Un *átomo* es un símbolo de relación aplicado sobre tantos términos como sea su aridad:

$$(R \in Rnes)\,\&\,(ar(R) = r)\,\&\,(t_1, \ldots, t_r \in Térm)$$
$$\Rightarrow\; R(t_1, \ldots, t_r) \in Atom$$

  Un átomo se dice *abierto* si en él aparece alguna variable. Las ocurrencias de variables en átomos abiertos se dicen ser *libres*. Un átomo es *cerrado* si no posee argumentos libres.

- Una *descripción* es un conjunto de átomos:

$$\phi_1, \ldots, \phi_q \in Atom \;\Rightarrow\; \{\phi_1, \ldots, \phi_q\} \in Desc$$

Las variables libres en átomos de una descripción aparecen también *libres* en la descripción

Ejemplo 2.3 (Gira de candidato). Consideremos una *gira electoral* de un *candidato* que ha de visitar muchas *poblaciones*, las cuales poseen al menos 10^4 *habitantes* y están en diversas *entidades federativas*. Las poblaciones o bien son *capitales* de sus respectivas entidades o bien son ciudades *interiores*. En cada ciudad hay un cierto número de *votos asegurados* siempre que el candidato visite la ciudad. Entre cualesquiera dos poblaciones hay una *distancia* entre ellas.

Dos ciudades en una misma entidad son *vecinas*.

Dos capitales de entidades *colindantes* también son *vecinas*.

El propósito del candidato es que estando en una ciudad *inicial* ha de arribar a una *destino*, viajando siempre de una ciudad a una vecina de esa ciudad, procurando minimizar la distancia y maximizar la cantidad de votos asegurados.

Formalicemos este problema sobre la signatura siguiente:

Constantes. La palabra *Candidato* es una constante

Para cada ciudad con más de 10^4 habitantes, el nombre de esa ciudad es una constante.

Para cada número que describa una distancia o una cantidad de votos, su representación decimal es una constante.

Variables. Consideraremos las que sean necesarias (acaso 10), con nombres $x, y, z, \ldots$, acaso con subíndices

Relaciones. Consideraremos las relaciones siguientes (con connotaciones evidentes):

- *Capi*(x): La ciudad x es una capital
- *Inte*(x): La ciudad x es interior
- *Voto*(x, z): z es el número de votos potencialmente asegurados en la ciudad x
- *Veci*(x, y): La ciudad x es vecina de la ciudad y
- *CpDe*(x, y): La ciudad x es la capital de la entidad donde está la ciudad y
- *Dist*(x, y, z): z es la distancia de la ciudad x a su vecina y
- *EsEn*(x): El candidato está en la ciudad x
- *DiRe*(z): El candidato ha recorrido en total hasta el momento la distancia z (medida en miles de kilómetros)
- *VoAs*(z): El candidato ha asegurado en total hasta el momento el número de votos z (medido en miles de votos)

Reglas. *Conm* y *Mism* que explicaremos más abajo

Acciones. Hay una única acción:

- *VaDA*(x, y): El candidato va de la ciudad x a la ciudad y

Como meros ejemplos de descripciones citemos a los siguientes:

1. "x es vecina de la ciudad y que a su vez es vecina de la capital z"

$$Desc_1 = \{\, Veci(x, y),\, Veci(y, z),\, Capi(z)\}$$

2. "x es vecina de la ciudad de *Guanajuato*, vecina ésta de *Querétaro*"

$$Desc_2 = \{\, Veci(x, Guanajuato),\, Veci(Guanajuato, Querétaro)\}$$

3. "El candidato ha recorrido 50 000 kms, está en Tijuana y Mexicali es vecina de Tijuana."

$$Desc_3 = \{EsEn(Tijuana),\, DiRe(50),\, Veci(Tijuana, Mexicali)\}$$

4. "La capital de la entidad en la que se encuentra Acapulco es vecina de la capital de la entidad en la que se encuentra Cuautla"

$$Desc_4 = \{CpDe(x, Acapulco), CpDe(y, Cuautla), Veci(x, y)\}$$

En cuanto a las reglas, cada regla ρ es del tipo *de Horn*: Tiene como antecedente un conjunto de átomos y como consecuente un átomo. Si

$$antecedente(\rho) = \{\phi_1, \ldots, \phi_k\}$$

y $consecuente(\rho) = \psi$ escribiremos

$$\rho = (\{\phi_1, \ldots, \phi_k\} \rightarrow \psi),$$

o bien

$$\rho : \frac{\phi_1, \ldots, \phi_k}{\psi}.$$

En el ejemplo 2.3 las reglas son las siguientes:

- "La relación de "vecino" es conmutativa": $Conm : \frac{Veci(x,y)}{Veci(y,x)}$.
- "Dos ciudades en una misma entidad, determinada por una capital, han de ser vecinos": $Mism : \frac{CpDe(x,y), CpDe(z,y)}{Veci(x,z)}$.

En cuanto a las acciones, cada acción tiene asociadas una *precondición* que es un conjunto de átomos y una *postcondición* que es un conjunto de átomos. A una acción se le puede ver como una regla "parametrizada" por las variables libres que aparezcan en ella. Si

$$precondición(A_i(\mathbf{x})) = \{\phi_1(\mathbf{x}), \ldots, \phi_k(\mathbf{x})\}$$

y $postcondición(A_i(\mathbf{x})) = \{\psi_1(\mathbf{x}), \ldots, \psi_l(\mathbf{x})\}$ escribiremos

$$A_i(\mathbf{x}) = (\{\phi_1, \ldots, \phi_k\} \Rightarrow \{\psi_1(\mathbf{x}), \ldots, \psi_l(\mathbf{x})\}),$$

o bien $A_i(\mathbf{x}) : \frac{\phi_1(\mathbf{x}),\ldots,\phi_k(\mathbf{x})}{\psi_1(\mathbf{x}),\ldots,\psi_l(\mathbf{x})}$.

En el ejemplo 2.3 la acción es la siguiente:

- "Estando el candidato en una población, puede ir a cualquiera de las ciudades vecinas. Hecho lo cual, el candidato estará, en efecto, en la ciudad a la que fue": $VaDA(x, y) : \frac{Veci(x,y), EsEn(x)}{EsEn(y), Veci(x,y)}$

Capítulo 3

Esquemas de localización de planes

3.1. Algunos métodos de resolución de problemas

En este capítulo revisaremos algunos métodos propios de la *inteligencia artificial* para resolver problemas. Este capítulo nos abrirá el panorama para la resolución del problema de planeación pero con el enfoque de autómatas finitos.

Método de generación y prueba. Este método se basa en dos módulos, uno que se encarga de generar todas las soluciones posibles y el otro módulo se encarga de evaluar cada solución propuesta, ya sea aceptándola o rechazándola.

Es posible que el módulo que genera produzca una salida con todas las posibles soluciones y después el módulo que prueba se encargue de evaluar cada solución. Es común que se aplique un método de intercalamiento el cual genera un cantidad fija de soluciones las cuales después se evalúan. Veamos un sencillo algoritmo en pseudo-código que resuelve el problema anterior:

- Mientras no se encuentre la solución o no se puedan generar más posibles soluciones

- genere una solución posible
- pruebe la solución posible

- si se encuentra una solución aceptable, dar la solución, en otro caso; dar mensaje de fracaso

Es común que el paradigma de generación de pruebas se utilicen para resolver problema de identificación. En los problemas de identificación se dice que el módulo generador antes visto produce una *hipótesis*. Para usar el paradigma de generación y prueba con el objeto a identificar, por ejemplo, un árbol, podríamos consultar un libro acerca de árboles, recorrer todas las páginas e ir comparando los esquemas para detectar cuál es el árbol buscado.

Veamos algunas características de los generadores:

- Son completos: en algún momento dado el generador produce todas las soluciones posibles
- No son redundantes: Deben estar hechos para reconocer cuándo se está generando una solución propuesta antes
- Usan información que limita las posibilidades de generar soluciones fuera del contexto

El método antes visto podemos definirlo también como un método de fuerza bruta, ya que se están generando todas las posibles soluciones una a una para poder llegar a la solución deseada. Veamos ahora el concepto de *Redes Semánticas.*

Cuando queremos resolver un problema computacional, normalmente necesitamos representar el problema de alguna manera. Por ejemplo, normalmente comenzamos representando el problema en español, el lenguaje español nos proporciona una manera eficaz de poder explicar el problema, pero para nuestros fines no es suficiente, pues necesitamos abarcar *explícitamente* todos los posibles casos de un problema, lo cual en español se torna un

tanto complicado. El lenguaje español es un lenguaje natural, por lo tanto da pie a muchas ambigüedades. Al tener ambigüedades en la definición del problema, seguramente tendremos problemas en el momento de la resolución del mismo pues se presentarán situaciones confusas ya que los lenguajes naturales no se adhieren a un conjunto de reglas tan estrictas como los lenguajes formales [1].

En general, un lenguaje natural es aquel que ha evolucionado con el paso del tiempo para fines de comunicación humana. Estos lenguajes continúan su evolución sin tomar en cuenta reglas gramaticales formales; cualquier regla se desarrolla después de que sucede el hecho, en un intento por explicar, y no determinar, la estructura del lenguaje. Como resultado de esto, pocas veces los lenguajes naturales se ajustan a reglas gramaticales sencillas u obvias.

En contraste con los lenguajes naturales, los formales están bien definidos por reglas preestablecidas, y por lo tanto, se ajustan con todo rigor a ellas. Como ejemplo tenemos los lenguajes de programación de computadoras, los lenguajes matemáticos y la lógica de proposiciones. Gracias a esta adhesión a las reglas, es posible construir compiladores o describir problemas como el que estamos tratando en esta tesis.

Es necesario contar con una forma de traducir el problema de las simples palabras a un lenguaje más formal. Por ejemplo un lenguaje matemático es conveniente pues es conocido por muchas personas.

En el ejemplo 2.3 pudimos describir el problema en español. Éste parece ser claro desde este punto de vista, y se utilizó también un lenguaje matemático para describirlo. Cuando se formalizó el problema fue cuando se encontraron las dificultades para expresarlo de una manera matemática.

La tarea de presentar un problema es muy importante, pues debe ser tal que cualquier persona que se interese en el problema pueda entenderlo y

describirlo en su totalidad. Una vez que un problema se describe mediante una buena representación, el problema está casi resuelto.

Definición 3.1. *Para nuestros fines, entenderemos por una representación un conjunto de convenciones sobre la forma de describir un problema.*

Una representación consiste en las siguientes partes fundamentales:

- Un alfabeto Σ
- Un conjunto de restricciones R que habla de cómo los símbolos de Σ pueden agruparse
- Una parte operativa, es decir, un conjunto de acciones válidas A que se pueden ejecutar durante el proceso de resolución del problema
- Una parte semántica S que estalece una forma de asociar el significado con las descripciones

Una *Red Semántica* desde el punto de vista léxico es una gráfica que contiene nodos y enlaces. Una red semántica se asemeja a los autómatas finitos vistos en el Capítulo 1. Nuestro objetivo es conocer diversas formas de atacar un problema de planeación para posteriormente en el siguiente capítulo establecer la solución al problema de planeación por medio de la técnica de autómatas finitos. La representación implicada en el ejemplo 2.3 es una red semántica.

Veamos otro ejemplo de resolución de problemas, *Reducción del problema.* En ocasiones es posible convertir una meta primaria en submetas fáciles de lograr. Cada submeta puede ser dividida a su vez en submetas de nivel inferior. Veamos un ejemplo.

Ejemplo 3.1 (Movimiento de objetos). Se tiene una mesa con sólo cubos, pirámides, esferas y una mano de robot. Se tiene el procedimiento MUEVE que resuelve problemas de manipulación de objetos. MUEVE trabaja bajo mandatos como el siguiente procedimiento:

$$Coloca < nombredeobjeto > sobre < otroobjeto >$$

Para poder ejecutar la acción, el procedimiento busca una sucesión de movimientos para el robot de una sola mano que toma un solo objeto a la vez. MUEVE consiste en procedimientos que reducen los problemas dados a otros más simples buscando así la solución del problema. Veamos algunos procedimientos que usa MUEVE:

- COLOCA pone un objeto encima de otro. Activa otros procedimientos que se encargan de encontrar un lugar adecuado en la cima del objeto destino
- CONSIGUE_ESPACIO encuentra un lugar en la cima de un objeto destino
- HAZ_ESPACIO ayuda al procedimiento anterior cuando es necesario, moviendo obstáculos hasta que haya espacio suficiente para el objeto en movimiento
- TOMA agarra objetos
- DESPEJA_CIMA limpia la cima del objeto seleccionado.
- DESHAZTE_DE quita los obstáculos poniéndolos sobre la mesa
- SUELTA hace que la mano del robot suelte lo que tiene agarrado
- MUEVE traslada objetos, un vez que han sido agarrados

Con base en las acciones anteriores se puede llegar a diferentes configuraciones de objetos dado un estado inicial de los objetos en una supuesta mesa. La idea de este ejemplo es tener un mecanismo de resolución de problemas, en este caso el de *reducción de problemas*. Es claro que para poder aplicar MUEVE éste se debe de encargar de verificar si hay espacio, si la cima está ocupada, etc. De esta manera se está descomponiendo el problema en metas más pequeñas. Un *árbol de metas*, como el que se muestra en la figura 3.1, es un árbol semántico en el que los nodos representan metas y las ramas indican la forma en que se pueden lograr las metas, mediante la solución de una o más submetas.

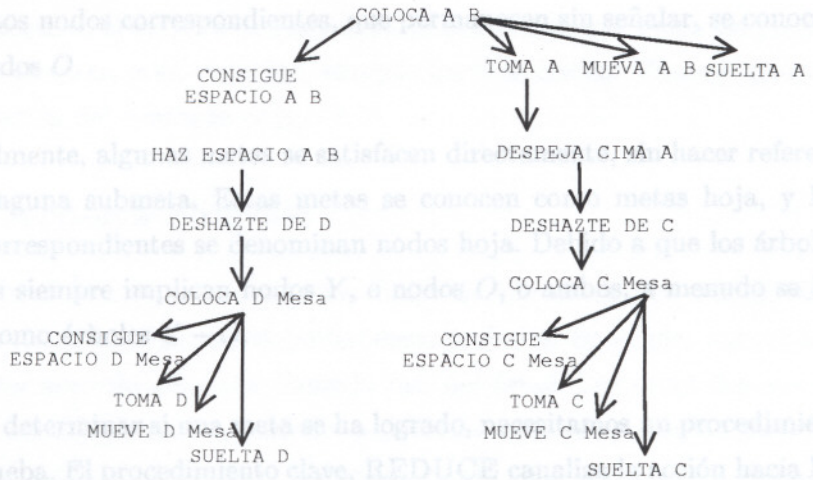

Figura 3.1: Árbol de metas

Los hijos de cada nodo corresponden a submetas inmediatas; cada padre de nodo corresponde a la supermeta inmediata. El nodo superior, que no tiene padre, es el nodo meta *raíz*.

Un árbol de metas hace transparente los complicados argumentos de MOVER. La acción de despejar la cima del cubo A se muestra como una submeta inmediata de tomar el cubo A. Despejar la cima del cubo A es también una submeta de colocar al cubo A en algún lugar encima del cubo B, pero no se trata de una submeta inmediata.

Todas las metas que se muestran en el ejemplo se satisfacen sólo cuando todas las submetas inmediatas quedan satisfechas. Las metas que se satisfacen sólo cuando todas sus submetas inmediatas quedan satisfechas se conocen como metas Y. Los nodos correspondientes son los nodos Y, y se les señala colocando arcos en sus ramas.

La mayoría de los árboles de metas contienen también metas O. Estas metas se satisfacen cuando cualesquiera submetas inmediatas quedan satis-

fechas. Los nodos correspondientes, que permanecen sin señalar, se conocen como nodos O.

Finalmente, algunas metas se satisfacen directamente, sin hacer referencia a ninguna submeta. Estas metas se conocen como metas hoja, y los nodos correspondientes se denominan nodos hoja. Debido a que los árboles de metas siempre implican nodos Y, o nodos O, o ambos, a menudo se les conoce como árboles $Y - O$.

Para determinar si una meta se ha logrado, necesitamos un procedimiento de prueba. El procedimiento clave, REDUCE canaliza la acción hacia los procedimientos REDUCE-Y y REDUCE-0:
Para determinar, mediante REDUCE, si se ha logrado una meta,

- determine si la meta se ha logrado sin recurrir a submetas:
    - si es así, notifique que la meta se ha satisfecho;
    - de otro modo, determine si la meta corresponde a una meta Y:
        - si es así, use el procedimiento REDUCE-Y para determinar si la meta se satisface;
        - de otro modo, use el procedimiento REDUCE-0 para determinar si la meta se satisface

Teniendo a mano REDUCE, REDUCE-Y y REDUCE-0 resulta sencillo probar un árbol $Y - O$ completo: sólo utilice REDUCE en el nodo raíz, permitiendo que los diferentes procesos se llamen entre sí, según se requiera, y así puedan abrirse paso por el árbol.

Desde el punto de vista de la programación, MOVER consiste en un conjunto de procedimientos especializados. Cada vez que un procedimiento especializado llama a otro, efectúa un paso de reducción del problema. De manera más general, siempre que un procedimiento llame a un subprocedimiento, se da un paso de reducción del problema. Por tanto, la reducción

del problema es el método de resolución de problemas que todos los programas muestran en gran cantidad, excepto los más cortos. Veamos ahora la formalización del concepto de árbol.

3.2. Árboles binarios

Un *árbol binario* es un conjunto finito de elementos que o está vacío o está dividido en tres subconjuntos desarticulados. El primer subconjunto contiene un solo elemento llamado *raíz* del árbol. Los otros dos son en sí mismos árboles binarios, llamados *subárboles izquierdo* y *derecho* del árbol original. Un subárbol izquierdo o derecho puede estar vacío. Cada elemento de un árbol binario se llama *nodo* del árbol.

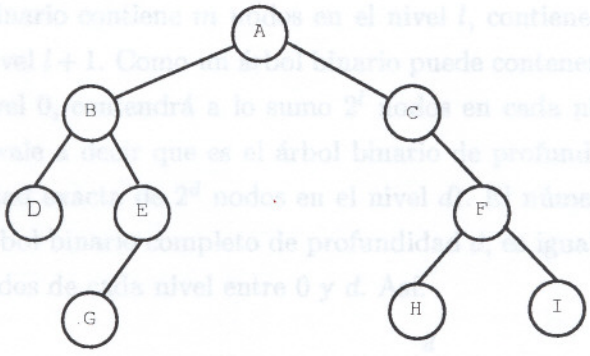

Figura 3.2: Árbol binario

En la figura 3.2 se muestra un método convencional de dibujar un árbol binario. Este árbol consiste de nueve nodos y tiene a A como raíz. Su subárbol izquierdo tiene a B como raíz y su subárbol derecho a C. Esto queda señalado por las dos ramas que salen de A: hacia B a la izquierda y hacia C a la derecha. La ausencia de ramas indica que es un subárbol vacío. Por ejemplo, tanto el subárbol izquierdo del árbol binario que tiene raíz en C como el subárbol derecho del árbol binario que tiene raíz en E están vacíos. Los árboles binarios con raíz en D, G, H, I tienen subárboles derecho e izquierdo vacíos.

Si un nodo que no es una hoja de un árbol binario tiene subárboles izquierdo y derecho no-vacíos, el árbol se llama *árbol estrictamente binario.* Así, un árbol estrictamente binario con n hojas tiene siempre $2n - 1$ nodos.

El *nivel* de un nodo de un árbol binario se define como sigue: La raíz del árbol tiene nivel 0 y el nivel de cualquier otro nodo del árbol es uno más que el nivel de su padre. Por ejemplo, en el árbol de la figura 3.2, el nodo E está en el nivel 2 y el H en el 3. La *profundidad* de un árbol binario es el máximo nivel de cualquier hoja del árbol. Esto es igual a la longitud del camino más largo de la raíz a cualquier hoja. Así, la profundidad d es el árbol estrictamente binario cuyas hojas están en el nivel d.

Si un árbol binario contiene m nodos en el nivel l, contiene a lo sumo $2m$ nodos en el nivel $l+1$. Como un árbol binario puede contener a lo sumo un nodo en el nivel 0, contendrá a lo sumo 2^l nodos en cada nivel l entre 0 y d. (Esto equivale a decir que es el árbol binario de profundidad d que contiene la cantidad exacta de 2^d nodos en el nivel d). El número total de nodos tn de un árbol binario completo de profundidad d, es igual a la suma del número de nodos de cada nivel entre 0 y d. Así:

$$tn = 2^0 + 2^1 + 2^2 + \cdots + 2^d = \sum_{j=0}^{d} 2^j$$

Por inducción, se puede mostrar que esta suma es igual a $2^{d+1} - 1$. Como todas las hojas de un árbol de este tipo están en el nivel d, el árbol contiene 2^d hojas y, en consecuencia, $2^d - 1$ nodos que no son hojas.

De manera análoga si se sabe el número de nodos, tn, de un árbol binario completo, se puede calcular su profundidad d, usando $tn = 2^{d+1} - 1$. Por lo tanto se puede decir que, en un árbol binario completo, $d = log_2(tn + 1)$. La importancia de un árbol binario completo es que es el árbol binario con el número de nodos máximo para una profundidad determinada. Dicho de otro modo, aunque un árbol binario completo contenga muchos nodos, la

distancia de la raíz a cualquier hoja es relativamente pequeña. Es importante hacer la analogía entre los árboles de metas y los árboles binarios, aunque en los árboles de metas no siempre son binarios, hay muchas similitudes entre ellos. Es bien sabido que las operaciones comunes en los árboles binarios involucran recursividad y en los árboles de metas también se necesitó de la recursivadad en nuestro procedimiento REDUCE.

Capítulo 4

Enfoque basado en autómatas finitos

4.1. Formalización

El campo de los sistemas de planeación se puede atacar de diferentes maneras. Los métodos más comunes para resolver el problema son eficientes pero carecen de *completez*. El método a tratar en este capítulo tiene ventajas y desventajas. Una desventaja es el tiempo que toma. Es decir, el método no es tan eficiente, pero a cambio nos devuelve algo muy valioso como es la completez, que se refiere al hecho de que dado un sistema de planeación siempre obtenemos la solución, lo cual no sucede con otros métodos.

El objetivo de este capítulo es resolver el problema de planeación por medio de la técnica de autómatas finitos. Veamos lo requisitos que se deben de tener para establecer un sistema de planeación:

- Un estado inicial del sistema de planeación
- Un conjunto de acciones válidas
- Un conjunto de estados que indican cuándo se llegó a la meta deseada

Básicamente los tres puntos anteriores es lo que se necesita para poder establecer un sistema de planeación. En el campo de los autómatas finitos, definimos un autómata finito como se vio en la definición 1.1.

Hagamos una correspondencia intuitiva entre lo que necesita un sistema de planeación y un autómata finito: El estado inicial del sistema de planeación que corresponda al $i \in S$ de los autómatas finitos, el conjunto de acciones válidas que corresponda a Σ de los autómatas finitos, el conjunto de estados que indican cuando se llegó a la meta deseada que sea $F \subseteq S$. De esta manera tenemos una correspondencia entre los autómatas finitos y los sistemas de planeación. De ahora en adelante cuando hablemos de sistemas de planeación podremos referirnos a los autómatas finitos. De hecho, preferiremos utilizar esta última opción.

Definición 4.1. *Decimos que un sistema de planeación P tiene una solución S cuando podemos encontrar una sucesión de acciones válidas A_n tales que la aplicación sucesiva de las acciones nos lleva a un estado de aceptación.*

El término *aplicar una acción* en un sistema de planeación corresponde a evaluar la función de transición del autómata finito en el estado actual del sistema con un elemento del alfabeto.
Ahora tenemos una idea general de la relación que existe entre los autómatas finitos y los sistemas de planeación.

4.2. Resolución del problema usando autómatas finitos

Sea P un sistema de planeación, nuestro objetivo es resolver el problema utilizando las técnicas de autómatas finitos. Nuestra finalidad es construir un autómata finito digamos M asociado a P. De ahora en adelante denotaremos por M_P al autómata finito asociado con P. Supongamos que tenemos un estado inicial i y un conjunto de acciones A_n nuestra tarea consiste en aplicar cada una de las acciones al estado inicial i. Para esto necesitamos saber si

las precondiciones de la acción A_i se encuentran contenidas en el estado i, entonces nuestra tarea inicial es verificar si

$$Precon(A_i) \subseteq i$$

Si es cierto lo anterior entonces es posible aplicar la acción A_i al estado inicial i. Procedemos de la siguiente forma

$$i' = Postcon(A_i) \cup \{i - \backslash Precon(A_i)\}$$

De esta manera hemos generado un nuevo estado i' el cual fue generado a partir del estado i. Si lo vemos en el contexto de los autómatas finitos tenemos que pasamos del estado i al estado i' por medio del símbolo A_i. De esta manera tenemos claro que el conjunto de estados S del autómata solución al principio estaría formado sólo por i ($S = \{i\}$). Entonces podemos formar un autómata M de la siguiente manera:

$$(S, \Sigma, \delta, i, F)$$

donde:

1. $S = \{i\}$
2. $\Sigma = \{A_n\}$
3. δ es la función que nos lleva de un estado a otro
4. i (un elemento de S) es el estado inicial y es igual al i que teníamos anteriormente
5. F (un subconjunto de S) es el conjunto de estados de aceptación, el cual es cuando el sistema de planeación se detiene

De esta manera hemos construido una relación entre los sistemas de planeación y los autómatas finitos, pero aún no hemos terminado. Tenemos una lista de tareas por cumplir para dejar claro la relación; para esto notemos lo siguiente. Antes de comenzar cualquier traslado de un problema de

planeación a autómatas finitos debemos estar seguros si acaso el problema tiene solución. Existen diversas maneras; en principio se tienen los problemas de planeación tradicionales, los cuales sabemos que tienen una solución. Enseguida tenemos los problemas que están abiertos y todavía no se prueba la existencia de alguna solución, estos problemas no serán de nuestro interés. Así tenemos localizados los problemas, una vez que tenemos un problema para el cual estamos seguros que hay al menos una solución, procedemos a *encontrar un plan.* Dicho plan, después de haber sido ejecutado debe de satisfacer las condiciones finales del problema. Después de esto diremos que hemos encontrado un *plan exitoso* o dicho en otras palabras hemos llegado a un estado final del autómata. Debemos notar que al haber encontrado un plan exitoso cabe la posibilidad de que existan más planes que nos lleven al estado final. Es posible aplicar el algoritmo de resolución del sistema y encontrar más planes que nos lleven al final. Es interesante pensar en el *plan más corto*, definiendo a éste como el plan que aplica menos acciones para poder llegar al estado final. De esta manera tenemos un método para poder escoger al plan más corto de un conjunto de planes.

Existen casos en el que dos planes pueden ser equivalentes. Es decir, ambos planes nos llevan al estado final y además por ejemplo, el costo de ambos planes para llegar al estado final es el mismo; si esto sucede, diremos que ambos planes son equivalentes. De esta manera podemos decir que los dos problemas de planeación son homomorfos. Tiene sentido hablar de homomorfismos entre problemas de planeación pues a cada problema de planeación le corresponde un autómata finito y en el campo de los autómatas finitos se tiene el concepto de homomorfismo. Podemos pensar en lo siguiente. Supongamos que tenemos un autómata finito M, y tenemos el lenguaje del autómata, digamos $L(M)$. Sabemos que el lenguaje del autómata es el conjunto de todas las cadenas (palabras) en Σ^* tales que al evaluar cada palabra en la función de transición llegamos a un estado final. Es interesante pensar en el conjunto de estados que son visitados al evaluar las palabras del lenguaje. Al conjunto de estados que son visitados al menos una ocasión

en la evaluación del lenguaje le llamaremos *Parte conexa* del autómata.

Existen ocasiones en las que un autómata contiene muchos estados, pero en el momento de calcular la parte conexa del autómata se reduce sorprendentemente. Es posible que un sistema de planeación se reduzca en gran medida cuando se calcula la parte conexa. Veamos un concepto muy importante en el campo de los autómatas finitos. En el capítulo uno probamos el teorema 1.4. *El Teorema de Kleene* [1] es muy importante, pues nos prueba que a cualquier expresión regular r le podemos asociar un autómata finito y por otro lado también vimos la *Regla de Arden* (Teorema 1.6, [1]) la cual nos garantiza que para todo autómata finito podemos construir una expresión regular asociada. De esta manera tenemos un par de herramientas muy poderosas. Veamos las consecuencias de estos dos resultados en el campo de los Sistemas de Planeación.

Ya hemos visto la manera de calcular el autómata finito asociado a un problema de planeación. Una vez que hemos calculado el AF por la *Regla de Arden* podemos encontrar una expresión regular que representa al lenguaje del problema de planeación. Sabemos que la expresión regular es una manera de representar TODAS las cadenas aceptadas por una autómata finito. De esta manera, al calcular la expresión regular asociada al problema de planeación tenemos una forma concisa de representar TODAS las soluciones al problema dado; esto es una consecuencia sorprendente.

Hemos comentado que existen ocasiones en que dos autómatas finitos pueden aceptar el mismo conjunto de cadenas. Esta implicación es clara pero enseguida mostraremos que existe un algoritmo para determinar si dos autómatas finitos aceptan el mismo conjunto.

Teorema 4.1. *El conjunto de cadenas aceptadas por un autómata finito M que tiene n estados es:*

1. *no vacío si y sólo si el autómata finito acepta una cadena de longitud*

menor a n

2. *infinito si y sólo si el autómata acepta alguna cadena de longitud l, en donde* $n \leq l \leq 2_n$

Así pues existe un algoritmo para determinar si un autómata finito acepta un número finito o infinito de cadenas o no acepta ninguna.

Demostración. La prueba del teorema se puede revisar en [1]. □

Teorema 4.2. *Existe un algoritmo para determinar si dos autómatas finitos son equivalentes (es decir, si aceptan el mismo lenguaje)*

Demostración. Sean M_1 y M_2 autómatas finitos que aceptan a L_1 y a L_2, respectivamente. Por lo anteriormente visto de las expresiones regulares $(L_1 \cap L_2^c) \cup (L_1^c \cap L_2)$ es aceptado por algún autómata finito, M_3. Es fácil ver que M_3 acepta una palabra si y sólo si L_1 es distinto de L_2. De aquí que, por el teorema anterior, exista un algoritmo para determinar si $L_1 = L_2$. □

4.3. Minimización de problemas de planeación

En el campo de los autómatas finitos es común hablar de la minimización de autómatas. Dado que tenemos una relación entre los autómatas y los sistemas de planeación podremos hablar de manera natural de la minimización de un problema de planeación. Recordemos lo anteriormente visto en el capítulo 1 acerca de las relaciones de equivalencia y clases de equivalencia. Podemos asociar con un lenguaje cualquiera L una relación de equivalencia natural R_L; es decir, xR_Ly si y sólo si para cada z, xz y yz están en L o no lo están. En el peor de los casos cada cadena está, por sí misma, en una clase de equivalencia, pero puede haber menos clases. En particular, el índice (número de clases de equivalencia) siempre es finito si L es un conjunto regular.

Existe también una relación de equivalencia natural sobre las cadenas asociadas con un autómata finito. Sea $M = (Q, \Sigma, \delta, q_0, F)$ un AFD. Para

x y y en Σ^* sea xR_MY si y sólo si $\delta(q_0, x) = \delta(q_0, y)$. La relación R_M es reflexiva, simétrica y transitiva, puesto que el símbolo '=' tiene tales propiedades, y por tanto R_M es una relación de equivalencia. R_M divide al conjunto Σ^* en clases de equivalencia, una por cada estado que se puede alcanzar desde q_0. Además, si xR_My, entonces xzR_Myz para toda z en Σ^*, ya que,

$$\delta(q_0, xz) = \delta(\delta(q_0, x), z) = \delta(\delta(q_0, y), z) = \delta(q_0, yz)$$

Una relación de equivalencia R tal que xRy implique $xzRyz$ se dice que es *invariante por la derecha (con respecto a la concatenación).* Vemos que cada autómata finito induce una relación de equivalencia invariante por la derecha, definida como se definió R_M sobre las cadenas de entrada. Este resultado se formalizará en el siguiente teorema. De esta manera podemos hablar de que cada problema de planeación induce una relación de equivalencia.

Teorema 4.3. *Las tres proposiciones siguientes son equivalentes:*

1. *El conjunto $L \subseteq \Sigma^*$ es aceptado por algún autómata finito*
2. *L es la unión de algunas clases de equivalencia de una relación de equivalencia invariante por la derecha de índice finito*
3. *Sea R_L la relación de equivalencia definida por: xR_Ly si y sólo si para toda z en Σ^*, xz está en L exactamente cuando yz está en L. Entonces R_L es de índice finito*

Demostración. (1) $\rightarrow$ (2) Supongamos que L es aceptado por algún *AFD* $M = (Q, \Sigma, \delta, q_0, F)$. Sea R_M la relación de equivalencia xR_My si y sólo si $\delta(q_0, x) = \delta(q_0, y)$. R_M es invariante por la derecha ya que, para cualquier z, si $\delta(q_0, x) = \delta(q_0, y)$, entonces $\delta(q_0, xz) = \delta(q_0, yz)$. El índice de R_M es finito, puesto que el índice es cuando mucho el número de estados de Q. Aun más, L es la unión de aquellas clases de equivalencia que incluyen una cadena x tal que $\delta(q_0, x)$ está en F, es decir, las clases de equivalencia correspondientes a los estados finales.

(2) $\rightarrow$ (3) Mostramos que cualquier relación de equivalencia E que satisfaga (2) es un *refinamiento* de R_L; esto es, cada clase de equivalencia de E está completamente contenida en alguna clase de equivalencia de R_L. Por consiguiente, el índice de R_L no puede ser mayor que el índice de E y, por tanto, es finito. Supongamos que xEy como E es invariante por la derecha, para cada $z \in \Sigma^*$, $xzRyz$, y por tanto $yz \in L$ si y sólo si $xz \in L$. Así pues, xR_Ly, y en consecuencia, la clase de equivalencia de $x \in E$ está contenida en la clase de equivalencia de $x \in R_L$. Conluimos que cada clase de equivalencia de E está contenida en alguna clase de equivalencia de R_L.

(3) $\rightarrow$ (1) Primero debemos mostrar que R_L es invariante por la derecha. Supongamos que xR_Ly y sea w una cadena en Σ^*. Debemos probar que xwR_Lyw; esto es, para cualquier z, xwz está en L exactamente cuando ywz está en L. Pero como xR_Ly, sabemos por la definición de R_L que para cualquier v, xv está en L exactamente cuando yv está en L. Hacemos $v = wz$ para probar que R_L es invariante por la derecha.

Ahora hagamos Q' el conjunto finito de clases de equivalencia de R_L y $[x]$ el elemento de Q' que contiene a x. Defínase $\delta'([x], a) = [xa]$. La definición es consistente, ya que R_L es invariante por la derecha. De haber escogido y en lugar de x de la clase de equivalencia $[x]$, hubiéramos obtenido $\delta'([x], a) = [ya]$. Pero xR_Ly, así que xz está en L exactamente cuando yz está en L. En particular, si $z = az'$, xaz' está en L exactamente cuando yaz' está en L, de modo que xaR_Lya y $[xa] = [ya]$. Sea$q_0' = [\epsilon]$ y $F' = \{[x] | x \in L\}$. El autómata finito $M' = (Q', \Sigma, \delta', q_0', F')$ acepta a L, puesto que $\delta'(q_0', x) = [x]$, y por tanto x está en $L(M')$ si y sólo si x está en F'. □

El teorema 4.3, entre otras consecuencias, básicamente nos dice que existe un *AFD* de estado mínimo único para cada conjunto regular.

Teorema 4.4. *El autómata de estado mínimo que acepta a un conjunto L es único salvo isomorfismo (es decir, un renombramiento de estados) y*

está dado como M' en la demostración del teorema anterior.

Demostración. En la demostración del teorema 4.3 vimos que cualquier AFD $M = (Q, \Sigma, \delta, q_0, F)$ que acepta a L define una relación de equivalencia que es un refinamiento de R_L. Por consiguiente el número de estados de M es mayor o igual que el número de estados de M' del teorema anterior. Si la igualdad es válida, entonces cada uno de los estados de M puede ser identificado con uno de los estados de M'. Esto es, sea q un estado de M. Debe de existir alguna $x \in \Sigma^*$, tal que $\delta(q, x) = q$, de otra manera q puede ser eliminada de Q, y hallar, así, un autómata más pequeño. Identifíquese a q con el estado $\delta(q_o, x)$ de M'. Esta identificación será consistente. Si $\delta(q_{0'}, x) = \delta(q_0, y)$, entonces, según la demostración del teorema anterior, x y y están en la misma clase de equivalencia de R_L. En consecuencia, $\delta'(q'_0, x) = \delta'(q'_0, y)$. □

Después de haber visto estos teoremas es interesante ver cómo los problemas de autómatas finitos son llevados al campo de los problemas de planeación. Notemos cómo los conceptos de relación de equivalencia, clases de equivalencia y autómatas mínimos son traducidos al campo de los problemas de planeación. En la siguiente sección veremos la aplicación de la teoría que hemos visto a ejemplos.

4.4. Algoritmo de solución

A continuación presentamos el algoritmo de solución acompañado de un ejemplo que ilustra el funcionamiento del algoritmo propuesto. Posteriormente, daremos una introducción del algoritmo. Como hemos venido comentando desde el principio, el método se basa en técnicas de autómatas finitos, por lo tanto nuestro objetivo será el de reducir el problema de planeación a uno de autómatas finitos.

La idea principal es la siguiente. Supongamos que tenemos un estado inicial descrito en un lenguaje formal como lo hemos visto anteriormete, también tenemos un conjunto de estados finales, un conjunto de acciones

válidas en el contexto del problema y por último un conjunto de predicados dan información acerca del problema en un momento dado. Debemos notar que para aplicar las acciones válidas éstas cuentan con un conjunto de precondiciones que se deben de cumplir para poder aplicar la acción. En otro caso, no será posible aplicarla.

Se comienza tomando el estado inicial y aplicándole una a una sus acciones que se puedan aplicar según lo siguiente: El conjunto de precondiciones que tiene el estado inicial debe de estar contenido en el conjunto de precondiciones que tiene la acción en turno. Al aplicar una acción se genera un nuevo estado del problema, esto es equivalente a lo visto en la figura 1.1, en nuestro caso el diagrama sería como el de la figura 4.1.

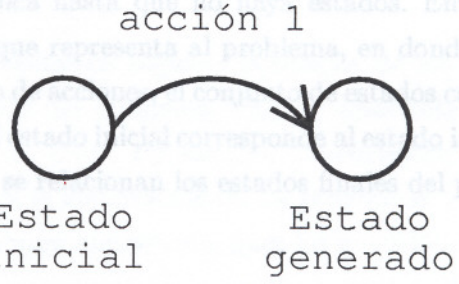

Figura 4.1: Estado generado apartir de aplicar la *acción 1*

El estado generado ahora forma parte del autómata resultado. El algoritmo, al generar el estado, se se asegura que éste no haya sido generado anteriormente por otra acción para evitar repeticiones y también verifica si acaso el estado nuevo es un estado final. Para esto, el conjunto de predicados del estado generado debe ser igual al conjunto de predicados de algún estado final de nuestro conjunto. Al aplicar una a una las acciones sobre el estado inicial se va llenando una pila con un conjunto de estados los cuales están en una pila de espera a ser tratados. En otras palabras, cada estado que se ingresa en la pila, se le deben de aplicar sus posibles acciones. Veamos ahora cómo sería la figura del estado inicial al aplicarle un conjunto de 4 acciones.

Éste quedaría como la de la figura 4.2

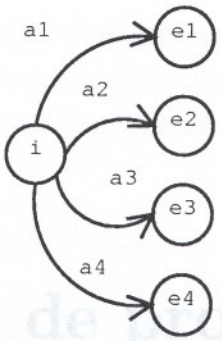

Figura 4.2: Diagrama del estado inicial al aplicarle 4 acciones

El algoritmo se aplica hasta que no haya estados. Entonces, al final, tenemos un autómata que representa al problema, en donde el alfabeto de entrada Σ es el conjunto de acciones, el conjunto de estados corresponde a los estados del problema, el estado inicial corresponde al estado inicial propuesto y los de estados finales se relacionan los estados finales del problema.

Capítulo 5

Presentación de programas

5.1. Preliminares

A continuación presentamos algunos bosquejos de los programas utilizados para la resolución del problema. Para el desarrollo del software se utilizaron los lenguajes *C, C++, Python [20].* La eficiencia de *C* y de *C++* han sido muy útiles para la elaboración de los programas y la introducción del lenguaje *Python* fue muy importante, pues los módulos de manipulación simbólica que contiene son muy poderosos. Gran parte del sistema se elaboró utilizando programación orientada a objetos, debido a su gran utilidad para el tratamiento de los estados, pues estos estados del autómata finito son vistos como objetos de software con propiedades y métodos propios. El tratamiento de las acciones también se efectúa por medio de objetos.

Los programas son de utilidad pues pueden ilustrar en algunos casos lo visto en la teoría. Es posible tener al final del programa un documento el cual nos da un seguimiento de cómo se fue ejecutando el algoritmo para llegar a la solución. Debemos notar que estos programas *no son completos* pues evidentemente funcionan con base en las capacidades de una computadora; es por eso que sólo debemos de tomarlos como una herramienta para ilustrar un poco lo antes visto en la teoría.

5.2. Un Ejemplo

Veamos un ejemplo muy ilustrativo del funcionamiento del algoritmo y de los programas que se implementaron. El ejemplo será muy sencillo, esto con la finalidad de tener un claro entendimiento del funcionamiento y no hacer complicada la lectura.

Ejemplo 5.1 (El juego de gato). Sea $\Sigma = \{O, X\}$ el alfabeto de entrada del problema. Sea $A = \{\text{PONE(O,X,Y)}, \text{PONE(X,X,Y)}\}$ el conjunto de acciones válidas. La primera acción coloca una O en la posición (x, y) y la segunda acción coloca una X en la posición (x, y). Sea $F = \{\text{ESTADOGANADOR}\}$ el estado al cual queremos llegar. El archivo de inicio es el siguiente:

9

$No\_Esta(1,1)$

$No\_Esta(1,2)$

$No\_Esta(1,3)$

$No\_Esta(2,1)$

$No\_Esta(2,3)$

$No\_Esta(2,3)$

$No\_Esta(3,1)$

$No\_Esta(3,2)$

$No\_Esta(3,3)$

El cual indica que el tablero esta completamente vacío en sus nueve casillas. Una típica acción es la siguiente: digamos que queremos colocar un símbolo en la casilla $(1,1)$, entonces ésta sería de la siguiente manera:

$\#Nombre$

$Pone(1,1)$

$$\#Precondiciones$$

$$No\_Esta(1,1)$$

$$\#Postcondiciones$$

$$Esta(1,1)$$

Esta acción indica que para poder aplicar la acción $Pone(1,1)$ se debe de cumplir el conjunto de precondiciones, en este caso es $No\_Esta(1,1)$. Al aplicarse la acción se genera un conjunto de postcondiciones que es $Esta(1,1)$, el cual indica que la casilla $(1,1)$ está ocupada. Veamos los pasos a seguir del algoritmo:

1. Insertar en la pila el estado inicial, es decir, insertar el tablero vacío a la pila
2. Mientras la pila no esté vacía hacer
   - Sacar la cima de la pila
   - Para cada acción de A, verificar si acaso se puede aplicar la acción. En caso afirmativo, aplicar la acción, generar un nuevo estado e introducirlo en la pila. En caso negativo, continuar con la siguiente acción
   - Al generar un nuevo estado, compararlo con el conjunto de estado finales, es decir, verificar si acaso el estado del tablero coincide con uno de los ganadores. En caso de que sea un estado final, el programa guarda el estado en un archivo y continúa
     - Anotar el movimiento en el archivo de salida AUTOMATA.TXT
     - Si el estado es final, anotarlo en el archivo FINALES.TXT
3. Reconstruir el autómata generado apartir del ciclo anterior
4. Imprimir el resultado en un archivo de salida.

5.3. Diseño

El sistema está compuesto por tres módulos los cuales están completamente escritos en lenguaje *Python*. Cada módulo se describe a continuación y un diagrama de bloques se presenta en la figura 5.1. El módulo *funciones.py* es el encargado del funcionamiento general. Éste para funcionar se alimenta de archivos de entrada que contienen el estado inicial, el conjunto de acciones y el conjunto de estados finales.

1. Módulo para tratar las acciones. *action.py*. Este módulo permite un tratamiento completo de las acciones. Se encarga de leer las acciones de los archivos, ejecutar una análisis sintáctico sobre el archivo para poder guardar los elementos en estructuras de datos adecuadas para su uso posterior. Este módulo es una clase. El constructor de la clase recibe como parámetro inicial el nombre de la acción que se va a cargar. Una vez cargado con éxito, entonces tenemos un objeto del tipo *Acción* que nos dará toda la información necesaria para aplicar el algoritmo de solución propuesto

2. Módulo para tratar los estados. *estado.py*. Este módulo, al igual que el de las acciones, nos genera un objeto del tipo *estado* el cual nos da funcionalidad e información acerca de los estados. Cada vez que generamos un estado del problema de planeación, se genera un objeto del tipo estado

3. Se implementó una pila clásica como la que se puede ver en [18] *stack.py*. Este módulo fue de gran utilidad para llevar un registro de los estados que habían sido generados y faltaban por tratar. El objetivo de esta pila es la de almacenar todos los estados que van siendo generados para posterior uso

4. Una clase que incluye parámetros *parametros.py*. Esta clase se utiliza para enviar los mensajes personalizados de cada problema que se vaya a tratar y parámetros internos del proceso

5. Programa principal *funciones.py*. Este incluye el algoritmo principal y hace uso de todas las clases anteriores. También hace uso de tres archivos de entrada que contienen el estado inicial, el conjunto de acciones y el conjunto de estados finales respectivamente.

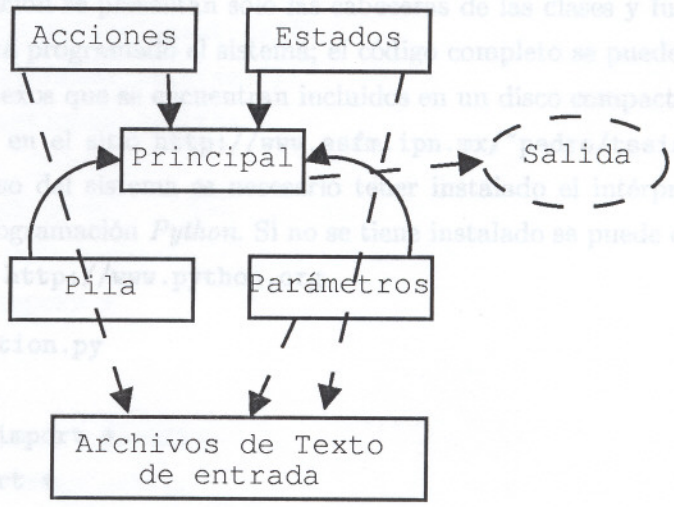

Figura 5.1: Diagrama de bloques del sistema

El sistema fue diseñado tomando en cuenta todos los átomos que se tienen durante el proceso de búsqueda de la solución. El diseño es completamente orientado a objetos; se trató de hacer todos los módulos lo más independientes los unos de los otros para poder tener un código reutilizable. Para efectos de los diferentes tipos de usuarios el sistema está diseñado para funcionar bajo el sistema operativo **Linux** y **Windows**. El sistema fue programado bajo la filosofía de desarrollo del software libre (Open source); es decir, cualquier persona interesada en el código lo puede obtener gratis del sitio `http://www.esfm.ipn.mx/~pedro/tesis`.
Para poder ejecutar el sistema es necesario cumplir ciertos requisitos de entrada

- Un archivo de texto que contiene el estado inicial

- Un archivo de texto por cada acción

Al finalizar el programa, éste nos devuelve un archivo de texto con un autómata finito en forma de tabla de transición el cual nos indica el desarrollo del problema.

A continuación se presentan sólo las cabeceras de las clases y funciones con las que está programado el sistema; el código completo se puede ver en los archivos anexos que se encuentran incluidos en un disco compacto junto con la tesis o en el sitio `http://www.esfm.ipn.mx/~pedro/tesis`. Para poder hacer uso del sistema es necesario tener instalado el intérprete del lenguaje de programación *Python*. Si no se tiene instalado se puede obtener gratis del sitio `http://www.python.org`

```
# Modulo: action.py

from string import *
from re import *

class Accion:
        def __init__(self, action):
        def readAccion(self):
        def getAll(self):
        def getAccion(self):
        def getNoPrecon(self):
        def getNoPostcon(self):
        def getPrecon(self):
        def getPostcon(self):
        def getTokens(self):

# Modulo estado.py

from string import *
```

```
class Estado:
        def __init__(self, state):
        def readState(self):
        def getAll(self):
        def getNoPredicates(self):
        def getPredicates(self):
        def getTokens(self):

# Modulo stack.py

from string import *

class Stack:
        def __init__(self):
        def getSize(self):
        def isEmpty(self):
        def getAllData(self):
        def push(self, key, value):
        def pop(self):
        def popCopy(self):
        def isElement(self, valor):


# Modulo funciones.py


from estado import *
from action import *
from stack import *
import sys
import os
```

```
from string import *

def iguales(L,M):
def getEdoInicial():
def isAccionValida(accion,estado):
def generaEstado(a,e, numero, pila):
def readNomAcciones():
def isTerminado(edoFinal, tmp):
def makeString(numero):
def automata(fuente, destino, accion):
# Inicia procedimiento principal
edo = 11
terminado = 0
edoFijo = 11
Acciones = readNomAcciones()
pila = Stack()
edoFinal = Estado('final.txt')
pila.push(edo, getEdoInicial) #inicializa la pila

while terminado != 1:
  llave = pila.pop().keys()[0]
    estadoActual = Estado(makeString(llave))
      edoFijo = edo
        for k in Acciones:
              accionActual = Accion(k)
              if isAccionValida(accionActual, estadoActual):
                  edo = edo + 1
                  tmp = generaEstado(accionActual, estadoActual, edo, pila)
                  if tmp != -1:
                      pila.push(edo, tmp)
                      noPaso = edoFijo + 1
                      print "destino = %s  fuente = %s" % (noPaso, edoFijo)
```

```
                automata(edoFijo, noPaso,  k)

                if isTerminado(edoFinal, tmp):
                    terminado = 1
                    print "Terminado!!!!!!!!!!!!"
            del accionActual
        del estadoActual
```

Conclusión

En los últimos años, el campo de los sistemas de planeación ha avanzado mucho. Hemos visto un método en el capítulo 2 que se basa en técnicas diferentes a la nuestra. El enfoque basado en autómatas finitos visto en esta tesis es un método que nos proporciona otra variante de resolución del problema, el cual constituye una variante a la encontrada normalmente en la literatura. El enfoque que seguimos no es más rápido que otros métodos propuestos, pero es efectivo en el sentido de que obtendremos no solamente una solución en algún momento, sino una descripción, como una expresión regular, de todas las posibles soluciones. Existen métodos sumamente conocidos y muy veloces, mas a costa de lograr sólo una solución, cuando tienen éxito, o incluso fallar en la localización de soluciones, aún cuando éstas existen. Estos métodos son eficientes, pero incompletos en muchos casos. Nuestro enfoque, en cambio, viene a ser completo y logran describir todas las posibles soluciones. Sin embargo, su complejidad computacional inherente puede volverlos prohibitivos en aplicaciones prácticas. Hemos procurado mostrar que los problemas de planeación modelados como autómatas finitos son Π-resolubles (en el sentido de localizar todas las posibles soluciones), bien que esto, en efecto, conlleva una complejidad computacional de orden mayor.

Bibliografía

[1] John E. Hopcroft, Jeffrey Ullman, *Introducción a la teoría de autómatas, lenguajes y computación*, CECSA,1997.

[2] J. Glenn Brookshear, *Teoría de la computación, lenguajes formales, autómatas y complejidad*, Addison-Wesley, 1989.

[3] N.J. Nilsson, *Principles of Artificial Intelligence.* Tioga Publ., Palo Alto, CA, 1980.

[4] Daniel S. Weld, *Recent Advances in AI Planning*, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350 USA. Technical Report UW-CSE-98-10-01; to appear in*AI Magazine*, 1999.

[5] Biplav Srivastava, Subbarao Kambhampati, *Synthesizing Customized Planners from Specifications*, Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, 1998.

[6] M. Ernst, T. Millstein, y D. Weld. *Automatic sat-compilation of planning problems.* In Proc. 15th Int. Joint Conf. AI, 1997.

[7] R. Fikes and N. Nilsson. *STRIPS: A new approach to the application of theorem proving to problem solving.* J. Artificial Intelligence, 1971

[8] R. Kambhampati, E. Lambrecht, y E. Parker. *Understanding and extending graphplan.* In Proc. 4th European Conference on Planning, Sep 1997.

[9] Subbarao Kambhampati. *Refinement planning as a unifying framework for plan synthesis.* AI Magazine 1997.

[10] J. Koehler. *Solving complex planning tasks through extraction of subproblems.* In Proc. 4th Intl. Conf. AI Planning Systems, Pittsburgh, PA, June 1998.

[11] A. L. Lansky. *Localized planning with action-based constraints.* J. Artificial Inteligence, September 1987.

[12] D. Smith y D. Weld. *Temporal graphplan.* Technical report, University of Washington, Dept. of Computer Science and Engineering, 1998.

[13] Dean, Thomas L., *Planning and Control*, Morgan Kaufmann Publishers Inc., 1991.

[14] Patrick Winston, *Inteligencia Artificial*, Addison-Wesley, 3a. Edición, 1994.

[15] Brian W. Kernighan, Dennis Ritchie, *El lenguaje de programación C*, Addison-Wesley, 2a. Edición, 1987.

[16] Bjarne Stroustrup, *El lenguaje de programación C++*, Addison-Wesley, 2a. Edición, 1998.

[17] Scott Robert Ladd, *C++ Techniques and Applications*, MyT Books, 1990.

[18] Aaron M. Tennenbaum, *Estructuras de datos en C*, Prentice Hall, 1a. Edición, 1993.

[19] Robert Sedgewick, *Algoritmos en C++*, Addison-Wesley, 1a. Edición, 1992.

[20] David M. Beazley *Python Essential Reference*, New Riders, 1a. Edición, 2000.