UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Métodos Heurísticos para Problemas de Optimización Combinatoria

T E S I S

QUE PARA OBTENER EL TÍTULO DE

ACTUARIO

P R E S E N T A :

SILVERIO DAVID MENESES BARAJAS

VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

M. en C. Virginia Abrín Batule
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
P r e s e n t e

Comunicamos a usted que hemos revisado el trabajo de Tesis:

" Métodos Heurísticos para problemas de Optimización Combinatoria"

realizado por Menoscs Barajas Silverio David

con número de cuenta 0719936_5 **, pasante de la carrera de** Actuaria'

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis	
Propietario	M. en I. María del Carmen Hernández Ayuso
Propietario	Lat. José Matías Alvarado Pentado
Propietario	Mat. Agustín Cano Garcés
Suplente	M. en I. María de Lu: Gasca Soto
Suplente	M. en I. Víctor Rafael Pérez Pérez

Consejo Departamental de Matemáticas

M.en C. ALEJANDRO BRAVO MOJICA

Indice

1. INTRODUCCIÓN

La conceptualización del término *heurística* tiene importantes interpretaciones y tiene sentido hablar de él cuando tenemos preguntas (problemas) cuya respuesta (secuencia finita de pasos) no podemos encontrar con suficiente rápidez, por la naturaleza de la pregunta.

Las interpretaciones que mencionaremos tienen su uso respectivo en *Investigación de Operaciones* (**IO**) e *Inteligencia Artificial* (**IA**), en la primera, el término se puede interpretar como *todo aquéllo que venga a la mente para resolver un problema*; mientras que en **IA** el sentido de heurística se utiliza a un problema lo podemos plantear mediante un *espacio de estados*, representado por una gráfica, donde cada vértice (estado) representa la evolución hacia la solución del problema por medio de *todo aquéllo que venga a la mente (por lo regular se usan operadores que modifican a los estados generando implícitamente a sus estados adyacentes hasta encontrar un estado que represente la solución)*; en este contexo la heurística es una estimación que nos indica qué tan promisoria sería la búsqueda en el espacio de estdos, si consideramos un estado determinado para alcanzar un estado que represente la solución del problema.

Es importante hacer notar que el término *instaucia* no tiene la debida correspondencia con su significado en español; sin embargo, por costumbre lo usaremos en lugar de *caso particular o ejemplo*.

La principal idea de este trabajo es relacionar los métodos de búsqueda heurísticos de **IA** como posibles métodos de solución para problemas que típicamente son estudiados en **IO**. Para esto, primero, se definiran claramente los conceptos de problema y algoritmo (con la relación natural que a cada problema le asigna un algoritmo que lo resuelva), para después, definir específicamente qué tipo de problemas nos interesará resolver; particularmente, aquéllos para los que no existe un algoritmo eficiente que los resuelva pues pertenecen a la clase de problemas conocidos como *No determinísticamente Polinomiales* (**NP**); esdecir, problemas en los que la cantidad de operaciones que tiene que realizar el algoritmo astronómica, lo que significa que no se podrá alcanzar en un tiempo razonable.

Los métodos que se utilizarán para solucionar este tipo de problemas fueron tomados de **IA** y se les conoce como *métodos de búsqueda heurísticos*, de los cuales el que sobresale es el A^*. Este método explota el posible conocimiento que se tenga del problema que se pretende resolver; además es un algoritmo que bajo circunstancias específicas es conocido teóricamente, lo que le da ventajas sobre los otros. En el **Capítulo II** se definen los conceptos de problema en general y problema de optimización combinatoria, además de darse la relación entre algoritmos y problemas. Luego se define con qué herramientas de la

Teoría de la Computación se representará a los problemas de optimización, estas herramientas se conocen como *esquemas de codificación* y son capaces de representar a un problema con un lenguaje formal determinado.

En el **Capítulo III**, se definen claramente los conceptos de *algoritmos determinísticos* y *no determinísticos*, los cuales implícitamente diferencian las clases de problemas. Además, en este capítulo se menciona la clase de problemas **NP-C** (a la cual pertenecen la mayoría de los problemas de **IO**) fundamentándose lo necesario para su estudio teórico.

En el **Capítulo IV**, se muestran inicialmente, tres estrategias para el diseño de algoritmos: *Divide y vencerás*, *Algoritmos Glotones* y *Búsqueda Local*; posteriormente se definen los problemas de búsqueda y las primeras estrategias para solucionarlos, conocidas en **IA** como de *fuerza bruta*: *Estrategia de Búsqueda a lo ancho* y *Estrategia de Búsqueda en profundidad*, las cuales se basan en los algoritmos de exploración en gráficas que se mencionan en el **Apéndice A**.

En el **Capítulo V**, después de que se mostró la necesidad de buscar en un espacio de estados, se analizarán los métodos de búsqueda heurísticos y se propondrán algunas funciones heurísticas que satisfagan las condiciones para garantizar la optimalidad del procedimiento A^*, pués también se verá que para algunos casos el procedimiento se comporta exponencialmente.

Silverio David Meneses.
Cd. Universitaria 1995.

2. PROBLEMAS Y ALGORITMOS

En este capítulo se analizará detalladamente el concepto de problema dandose dos puntos de vista clásicos: la de *[Garey, Johnson (1979)]* y la *[McNaughton(1982)]*, posteriormente, se definirán los problemas de Optimización Combinatoria y se dará una introducción breve al concepto de algoritmo. Finalmente se definirán a los Problemas de Decisión y los esquemas de codificación, estos últimos nos servirán para representar las instanscias de los problemas de decisión como lenguajes formales sobre algún alfabeto particular.

2.1 Conceptos Básicos

Un *Problema* Π, según *[Garey, Johnson (1979)]* es una *pregunta general*, normalmente con varios *parámetros (o variables libres)* que no tienen valores específicos.

Un problema se describe dando:

1. Una descripción general de todos sus parámetros.
2. Un enunciado especificando las propiedades que debe satisfacer la respuesta o solución.

Una *Instancia* del problema Π se obtendrá especificando valores particulares para todos los parámetros.

Sin embargo, *[McNaughton(1982)]* da una definición en la que no se consideran explícitamente los parámetros. (o incluso variables)

Definición 1 *Un problema* Π *es una clase de preguntas, donde cada pregunta de la clase es una instancia del problema.*

Generalmente, un problema es una clase finita de preguntas que pueden expresarse por una oración interrogativa. Por ejemplo, consideremos una colección C de conjuntos finitos y un entero positivo $k \leq \mid C \mid$, ¿Cúal es el entero positivo k, tal que C contiene al menos k subconjuntos ajenos?. En este caso la oración

el entero positivo k indica que la pregunta está determinada sólo cuándo se especifica un entero positivo. Ahora bien, si tomamos

$$C = A_1 \cup A_2 \cup A_3 \cup ... \cup A_n$$

con $A_i \neq \emptyset$ y $k = 100 \leq n$, tenemos la siguiente pregunta: ¿C contiene al menos 100 subconjuntos mutuamente ajenos?, dicha pregunta representa una instancia para el problema (a este problema se le conoce como *set packing*)

McNaughton clasifica a los problemas de matemáticas en dos tipos: los primeros son los que obtienen algún valor de una función que es evaluada por un argumento dado, donde no siempre la función es numérica. El segundo tipo de problemas es aquel donde las posibles respuestas pueden ser *si* o *no*.

La clasificación de McNaughton hace referencia a los problemas de decisión -que veremos más adelante- los cuales de cierta manera pueden representar a problemas que están en la primera clase.

El tipo de problemas que nos interesarán son los *Problemas de Optimización Combinatoria* (**POC**), que pertenencen a la primera clase de la clasificación de McNaughton. Y *[Papadimitriou, Steiglitz(1982)]* los definen como:

Definición 2 *Un problema de optimización combinatoria es un problema de maximización o minimización y se especifica por un conjunto de instancias, donde una instancia es una pareja* (Γ, k)*, donde el espacio de soluciones* Γ *denota al conjunto finito de todas las posibles soluciones y una función de costo* k*, que a cada solución le asocia un número real, definida como:*

$$k : \Gamma \rightarrow \Re$$

donde, de manera usual $\Re$ *denota al conjunto de los números reales.*

Denotaremos a $\gamma^* \in \Gamma$ *como la solución óptima global y en caso de minimización* γ^* *debe cumplir:*

$$k(\gamma^*) \leq k(\gamma), \forall \gamma \in \Gamma$$

En el caso de maximización, γ^* *debe cumplir:*

$$k(\gamma^*) \geq k(\gamma), \forall \gamma \in \Gamma$$

Considerando lo anterior, la pregunta natural que surge es ¿cómo se pueden resolver esos problemas? Es decir, ¿Existe un método que resuelva el problema para cualquier instancia de éste y además cada vez que consideremos la misma

instancia nos de el mismo resultado?.

En general a un método que sarisfaga los requerimientos de la pregunta anterior lo denominaremos *Algoritmo*; este concepto se formalizará más adelante.

De esta manera se puede pensar que un algoritmo es un procedimiento computacional bien definido que toma algún conjunto de valores como entrada y produce algún valor o conjunto de valores como salida, ver *[Rawlins(1991)]*. Además, debe dar la solución a problemas más generales y tener una estructura que se pueda modificar fácilmente. Por lo que un algortimo debe cumplir con las siguientes propiedades *[Rawlins(1991)]*:

- *Entrada:* Se debe especificar la información necesaria de la instancia
- *Salida:* Especifica los resultados obtenidos para una instancia
- *Definido:* Cada paso estará bien definido.
- *Finito:* Puede ser descrito en un número finito de pasos.
- *Correcto:* Debe encontrar la respuesta correcta del problema.
- *Completo:* El algoritmo termina.
- *Predecible:* Siempre encuentra el mismo resultado si se le da la misma entrada.

También existen otras propiedades que considerar cuando el algoritmo se implanta en algún lenguaje de programación:

- *Factibilidad:* El algoritmo debe ser realizable.
- *Claridad:* Los algoritmos claros son fáciles de codificar y de probar que son correctos.
- *Brevedad:* Los algoritmos cortos representan menos código, y confianza en la demostración de su desempeño.

Continuando con los conceptos de McNaughton, considérese la siguiente definición.

Definición 3 *Un algoritmo para un problema* Π, *es un conjunto organizado de instrucciones que responden una pregunta planteada, la cual es una instancia de* Π, *sujeto a las siguientes condiciones:*

1. *El algoritmo está realmente escrito (es posible que empiece a ser escrito como una expresión A finita de algún lenguaje).*

2. *Dada una entrada la ejecución del algoritmo deberá responder la pregunta del problema que se pretende responder con este.*

3. *Es posible observar la ejecución de un algoritmo como un proceso que se realiza paso a paso, donde las acciones que se realizan de un paso a otro son simples.*

4. *La acción de cada paso, incluyendo posiblemente una acción para terminar y los resultados de esta acción, con respecto a la ejecución, son estrictamente determinadas por la escritura de una expresión A, a partir de las entradas y el resultado de los pasos anteriores.*

5. *Después del término la respuesta se especifica claramente.*

6. *Para todos los posibles valores de las entradas, la ejecución finalizará después de un número finito de pasos.*

La idea de *Algoritmo Eficiente* siempre dependerá del tipo de problema que se esté analizando, de esta manera un problema puede ser solucionado usando diferentes algoritmos, de los cuales algunos pueden ser más *eficientes* que otros con respecto al *tiempo de ejecución*, que es el tiempo que tarda en resolver el problema, y al *espacio* que se ocupe.

Se supondrá que el *tiempo de ejecución* de un algortimo es proporcional al número de operaciones elementales que realiza al ejecutarse (llamado también *desempeño Computacional*). Llamaremos *tiempo* o *complejidad temporal de un algoritmo* a su desempeño computacional. Así mismo, se le llamará *espacio de almacenamiento* al número de localidades elementales de almacenamiento que utiliza.

El desempeño computacional y el espacio de almacenamiento dependen de la cantidad de datos de entrada (llamado también tamaño del problema).

Además se ha hecho una distinción que facilita la caracterización con respecto al desempeño computacional de los algoritmos, la cual los clasifica como *polinomiales* y *exponenciales* en función al tamaño de la *entrada*.

Lo anterior será más claro si consideramos la siguiente definicion, que representa el crecimiento asintótico de una función $t(n)$, donde $\mathbf{O}(s(n))$ denota la clase de funciones cuya función asintótica es $c \cdot s(n)$ para alguna $c \in \Re$, usualmente a $\mathbf{O}(s(n))$ se le conoce como el *Orden de complejidad s*.

Definición 4 *Diremos que f es de Orden de Complejidad s con respecto a la entrada de tamaño n y lo denotaremos como*

$$t(n) \in \mathbf{O}(s(n))$$

si $\exists c \in \Re$ *tal que*

$$t(n) < c \cdot s(n).$$

De tal manera, los algoritmos de tiempo polinomial se definen como aquéllos cuya función de complejidad es $\mathbf{O}(p(n))$, para algún polinomio p, donde n es el tamaño de la instancia del problema; $\mathbf{O}(s(n))$ representa sólo el *peor de los casos* del comportamiento del algoritmo para la instancia de tamaño n.

Cualquier algoritmo cuya función de complejidad no pueda ser acotada de esta manera se llamará *algoritmo de tiempo exponencial.* (esta definición incluye funciones de complejidad no polinomiales como $n^{\log n}$).

La distinción entre los dos tipos de algoritmos tiene un significado particular cuando se considera la solución de instancias grandes, pues se sabe que la razón de crecimiento de las funciones exponenciales es mucho mayor que en las polinomiales.

Esto hace pensar que los algoritmos de tiempo exponencial no son buenos (considerando sólo el peor de los casos). Sin embargo, un ejemplo que lo contradice es el *algoritmo simplex*, el cual, teóricamente, en el peor de los casos es de tiempo exponencial pero en la práctica es un algoritmo de tiempo polinomial, ver *[Fang,Puthenpura (1993)]*.

De tal forma que acordaremos que un problema no está *bien resuelto* hasta que no se conozca un algoritmo de tiempo polinomial que lo resuelva. Por lo tanto, se considerará un problema como *intratable* si el mejor algoritmo conocido que lo resuelve lo hace en tiempo exponencial.

2.2 Problemas de Decisión

Para continuar, consideremos ahora el segundo tipo de problemas de la clasificación de McNaughton, los cuales nos servirán para formalizar la Teoría de la Complejidad. A estos problemas se les conoce como *problemas de decisión* (**pd**) y sólo tiene dos posibles respuestas: la respuesta *sí* o la respuesta *no*.

La forma usual que se considerará para la especificación de problemas consiste de dos partes: la primera parte especifica una *Instancia Genérica* del problema en términos de varios componentes, los cuales pueden ser conjuntos, gráficas, funciones y números, entre otros. La segunda parte responde una *pregunta del tipo sí-no* en términos de la instancia genérica.

Para fijar ideas, consideremos como ejemplo el Problema del Agente Viajero. *[Garey, Johnson (1979)].*

Problema 1 *PROBLEMA DEL AGENTE VIAJERO* **(pav)**

Sea un conjunto finito de ciudades

$$C = \{c_1, c_2, c_3, \ldots, c_n\},$$

d una función definida como:

$$d : C \times C \to \mathbb{R}$$
$$(u, v) \mapsto d(u, v)$$

y además una constante B en el conjunto de los números enteros $\mathbf{Z}$.

Pregunta *¿Existe un recorrido que pase por todas las ciudades una y sólo una vez, cuya distancia sea menor o igual a B ?*
Es decir, ¿Existe un conjunto ordenado de ciudades de la forma

$$\langle c_{p(1)}, c_{p(2)}, c_{p(3)}, \ldots, c_{p(n)} \rangle$$

tal que

$$\sum_{i=1}^{n-1} d(c_{p(i)}, c_{p(i+1)}) + d(c_{p(n)}, c_{p(1)}) \leq B?$$

Este ejemplo en particular muestra cómo de un problema de optimización se puede derivar un problema de decisión.

Si en el problema de optimización se pregunta por una estructura de cierto tipo, que tenga un *costo mínimo* de entre todas las demás estructuras que lo solucionan, se puede asociar a este problema el problema de decisión que incluya una cota numérica B, como un parámetro adicional y que pregunte si existe una estructura del tipo requerido con un costo no mayor a B. (Los problemas de decisión también se pueden construir de problemas de maximización en forma análoga, reemplazando *no más* por *al menos*).

Por lo dicho en el parrafo anterior, nos interesara poder representar a los problemas de decisión de manera conveniente para su estudio teórico.

La representación que haremos de los problemas de decisión se basa en que la correspondencia entre un *lenguaje formal* y un problema de decisión está

dada por *los esquemas de codificación*, los cuales se usan para representar las instancias de un problema.

Para tener una idea clara de la correspondencia de los problemas con los lenguajes formales y los esquemas de codificación, ciertos términos deben estar bien especificados.

Epezaremos diciendo que un símbolo es un término primitivo que forma parte de un alfabeto.

Un alfabeto Σ es un conjunto finito de símbolos. Una cadena es una sucesión finita de elementos de Σ. Si x y y son cadenas, entonces la concatenación de x con y es la cadena xy. Ahora bien, si xyz es una cadena, entonces la subcadena x es el *prefijo*, y es una *subcadena* y la subcadena z es el *sufijo*; a la cadena vacía la representaremos por ε. Se denotará la longitud de la cadena x como $|\, x\,|$, e indicará el número de símbolos que tiene x. La longitud de la cadena vacía está dada por $|\, \varepsilon\,| = 0$.

Un *lenguaje* sobre Σ es un conjunto de cadenas formadas por símbolos de Σ. Sean L_1 y L_2 dos lenguajes, el lenguaje $L_1 \cdot L_2$, llamado sl la concatenación de L_1 *con* L_2 es el conjunto

$$L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Explicaremos ahora otro concepto importante; sea L un lenguaje, entonces definimos $L_0 = \{\varepsilon\}$ y $L_i = L \cdot L_{i-1}$ para $i = 1, 2, \ldots$. La *Cerradura de Kleene* de L se denota por $L^* = \bigcup_{i=0} L_i$. Es decir, L^* es el conjunto de todas las cadenas finitas que se pueden formar a partir de L.

Ahora bien, si $L \subset \Sigma^*$ diremos que L es un *Lenguaje formal* sonbre el alfabeto Σ

Por ejemplo, si $L = \{00, 11\}$ entonces

$$L^* = \left\{ \begin{array}{l} \varepsilon, 00, 11, 0000, 1100, 1111, 0011, 000000, 110000, \\ 111100, 000011, 000011, 110011, 111111, 001111, \ldots \end{array} \right\}.$$

De tal manera que L^* es el conjunto de todas las cadenas que tienen sucesiones de longitud par de ceros y/o unos y viceversa, incluyendo ε; por ejemplo, las cadenas $1010, 0101 \notin L^*$, porque sus sucesiones de ceros y/o unos son de longitud impar.

De la misma manera si Σ es un alfabeto, Σ^* es el conjunto de todas las cadenas de símbolos de Σ.

Por ejemplo, si $\Sigma = \{0, 1\}$, entonces Σ^* consiste de la cadena vacía ε, las cadenas 0, 1, 00, 11, 000, 001 y las cadenas finitas que se puedan formar con combinaciones de ceros y unos.

La correspondencia entre un lenguaje y un problema de decisión está dado por *los esquemas de codificación*, los cuales se usan para representar las instancias de un problema.

En general, un esquema de codificación para un problema de decisión Π, se utiliza para describir cada instancia de Π, con una cadena apropiada de símbolos sobre algún alfabeto $L \subseteq \Sigma^*$ fijo.

Entonces, como los esquemas de codificación son capaces de distinguir las instancias de un problema, con base en las cadenas de algun lenguaje L, necesariamente generan una partición (relación de equivalencia) sobre L, cuyas clases se distinguen como:

1. Aquéllas que no son codificaciones de instancias de Π.
2. Aquéllas que codifican instancias cuya respuesta es *no*.
3. Aquéllas que codifican instancias cuya respuesta es *sí*. A esta clase la denotaremos como Y_Π.

Esta tercera clase de cadenas representa un lenguaje asociado con el problema Π y el esquema de codificación **e**, y está dado por:

$$L[\Pi, \mathbf{e}] = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} \Sigma \text{ es un alfabeto usado por } \mathbf{e} \text{ y } x \text{ es una} \\ \text{codificacion bajo } \mathbf{e} \text{ de una instancia } I \in Y_\Pi \end{array} \right\}$$

Es este lenguaje formal el que nos servirá para poder representar todas las instancias válidas de un problema Π, es decir, si algún resultado se cumple para el lenguaje $L[\Pi, \mathbf{e}]$, entonces se cumple para el problema Π bajo el esquema **e**.

Sin embargo, el principal problema que se encuentra en esta parte es ¿cómo podemos construir un esquema de codificación para un problema Π, que sea simple (considerando el número de símbolos de un posible alfabeto)?. La respuesta a esta cuestión se puede encontrar si somos capaces de hacer que un esquema de codificación asocie instancias de Π con *cadenas estructuradas* de un alfabeto definido por $\Psi = \{1, 0, -, (,), [,]\}$ de la siguiente manera:

1. La representación binaria de un entero k como una cadena de ceros y unos (precedidos por $-$ si $k < 0$) es una cadena estructurada para k
2. Si x es una cadena estructurada para $k \in \mathbf{N}$, entonces $[k]$ es una cadena estructurada, que puede ser usada como un nombre (por ejemplo para un vértice de una gráfica o algún elemento de un conjunto)
3. Si $x_1, x_2, x_3, ..., x_m$ son cadenas estructuradas que representan a los objetos $X_1, X_2, X_3, ...X_m$, entonces $(x_1, x_2, x_3, ..., x_m)$ es una cadena estructurada que representa a la secuencia $<X_1, X_2, X_3, ..., X_m>$

Para encontrar un esquema de codificación para un **pd**, con el lenguaje especificado en el formato anterior, primero, se tiene que especificar cómo se construye la representación para cada tipo de objeto. Después, se tiene que construir una representación para cada objeto en la instancia como una cadena estructurada. La representación de la instancia completa está determinada por la *regla 3.*

Para hacer más explícita la relación entre los lenguajes y los problemas haremos una representación, a la que le llamaremos la representación usual de un problema. En general haremos las siguientes suposiciones:

1. Consideraremos la representación en decimal de los enteros (o como se menciono antes la representación binaria, según sea conveniente)

2. Los vértices de una gráfica los tendrán representados por los enteros $1, 2, 3, 4, ..., n$, codificados según la suposición 1. Una arista será representada por la cadena (i, j) donde i y j son las representaciones de los vértices.

3. Las expresiones lógicas con n variables proposicionales se representarán por cadenas en las cuales los símbolos:
   a). * representa la *conjunción*;
   b). + representa la *disyunción*;
   c). ¬ la negación;
   d). los paréntesis),(serán usados si es necesario enfatizar algún parámetro y
   e). los los enteros $1, 2, 3, 4, .., n$ representan las variables proposicionales.

Problema 2 *PROBLEMA DEL CLIQUE* (**pc**)
Sea una G una gráfica no dirigida. Un k-clique *es una subgráfica completa de G con k vértices, donde cada pareja de vértices están conectados por una arista. La question es:*

¿dada una gráfica G y un entero k, G contiene un k-clique?

Supongamos como instancia la gráfica de la figura 2-1, y $k = 3$, la cual puede ser codificada por la cadena:

$$3(1,2)(1,4)(2,3)(2,4)(3,4)(3,5)(4,5)$$

El primer entero representa el valor de k y las siguientes parejas representan aristas, donde v_i es representado por i.

El lenguaje L que representa el problema de clique es el conjunto de cadenas de la forma

$$k(i_1, j_1)(i_2, j_2)(i_3, j_3)...(i_m, j_m)$$

tal que la gráfica con aristas (i_r, j_r) para $1 \leq r \leq m$ tiene un k-clique.

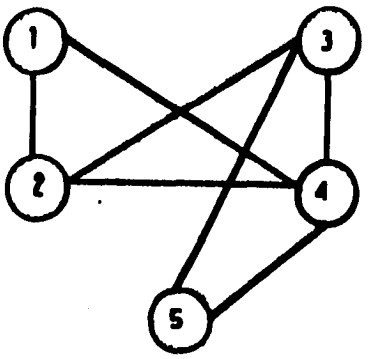


figura 2-1 Una gráfica no dirigida

Otros lenguajes también son capaces de representar el problema de clique, por ejemplo, la constante k puede estar al final de la representación de la gráfica, o en también se podría usar la representación decimal para los enteros.

En el siguiente capítulo será más clara la correspondencia que guardan los esquemas de codificación y los lenguajes con los problemas de decisión, además se usaran para definir el concepto de algoritmo, así como las diferentes clases de problemas: *polinomiales* (**P**), *no determinísticamente polinomiales* (**NP**) y *no determinísticamente polinomiales completos* (**NP-C**).

3. TEORIA DE LA COMPLEJIDAD

La idea principal de este capítulo es dar las bases suficientes para poder sustentar formalmente el concepto de algoritmo y la teoría de las clases de problemas. Empezaremos definiendo de manera general el concepto más importante para la representación de un algoritmo, la cual se conoce como *máquina de Turing* (**MT**). Continuaremos con la descripción de dos modelos equivalentes de la **MT** propuestos por *[Garey, Johnson (1979)]*, que nos servirán para definir las clases de problemas *polinomiales* (**P**), *no determinísticamente polinomiales* (**NP**) y *no determinísticamente polinomiales completos* (**NP-C**). Finalmente mencionaremos algunos problemas de optimización combinatoria que pertenecen a esta última clase.

3.1 Máquina de Turing

La máquina de Turing es una estructura matemática que fue propuesta por Alan Turing en 1936, *[Brookshear (1989)]*.

El propósito de su construcción fue desarrollar una estructura matemática en la cual se pudiera modelar cualquier proceso que pudiera considerarse como cálculo.

El modelo propuesto por Turing es más general que las computadoras actuales, porque una **MT** no está restringida por espacio de almacenamiento (esto se verá más adelante), mientras que la computadora puede agotar su capacidad de almacenamiento. Además la *Tesis de Turing* que se refiere a que el poder de cálculo de una **MT** es tan grande como el de cualquier posible sistema de computo, en general es aceptada por los especialistas de ciencias de la computación, ver *[Brookshear (1989)]*.

A continuación se dará una definición de una **MT-determinística (MTD)** tomada de *[Papadimitriou (1994)]*, la cual se especifica como sigue:

Definición 5 *Una máquina de Turing determinística es una cuarteta*

$$M = (Q, \Sigma, \delta, q_0)$$

donde:

1. Q *es un conjunto finito cuyos elementos son denominados estados*
2. $q_0 \in Q$ *es el estado inicial*

3. *Σ es un conjunto finito de símbolos y se llama alfabeto de la máquina M. Además Σ contiene a los elementos especiales b y $\diamond$; el símbolo blanco y el símbolo inicial, respectivamente.*

4. *Una función de transición*

$$\delta : Q \times \Sigma \to (Q \cup \{q_Y, q_N\}) \times \Sigma \times \{\rightarrow, \leftarrow, -\}$$

Donde q_Y es el estado de aceptación y q_N es el estado de rechazo, además las direcciones del cursor se especifican como $\rightarrow$ para derecha, $\leftarrow$ para izquierda y $-$ para movimiento nulo. *Se supondrá también que los símbolos $\rightarrow, \leftarrow, - \notin \Sigma \cup Q$.*

Generalmente la función δ es el *programa* de la máquina, y especifica para cada pareja de *estado-símbolo* una y sólo una tripleta de la siguiente manera

$$\delta(q, \sigma) = (q', \rho, D)$$

donde q' es algún estado, ρ es el símbolo que sustituirá a σ y $D \in \{\rightarrow, \leftarrow, -\}$ es la dirección en la cual el cursor se moverá.

Para las parejas de la forma $(q, \diamond)$ se deberá cumplir que

$$\delta(q, \diamond) = (q', \diamond, \rightarrow)$$

es decir, $\diamond$ siempre dirige el cursor a la derecha y nunca es borrado. Diremos también que *δ es una función de transición determinística.* Más concretamente a M se le llamará *máquina de Turing determinística.*

La ejecución de la máquina M empieza con el estado q_0, se inicializa una cadena finita de símbolos $w \in (\Sigma - \{b\})^*$ precedida del símbolo inicial $\diamond$. Diremos que w es la *entrada de la máquina.*

De esta manera siempre tendremos una configuración inicial para δ, la cual se especifica como

$$\delta(q_0, \diamond) = (q', \diamond, \rightarrow) \qquad (1)$$

A partir de esta configuración inicial, M realiza las operaciones que correspondan con la definición de δ. Como δ es una función completamente definida y el cursor nunca regresa a la izquierda, siempre empieza con *(1)*, por lo que sólo hay una razón para que la ejecución del programa termine: que se encuentre uno de los dos estados de paro, q_Y o q_N.

Si la máquina M termina y encuentra el estado q_Y se dirá que la máquina *acepta la entrada*, si el estado que se encontró fue q_N se dirá que *la entrada se rechaza.* También existe la posibilidad de que para algunas entradas la máquina M no se detenga.

Las máquinas de Turing pueden ser pensadas como algoritmos que resuelven problemas expresados con cadenas de símbolos que representan instancias de cualquier problema. (bajo algún esquema de codificación)

El modelo que a continuación se especifica, tiene como base el concepto de ***no determinismo***, el cual tiene una interpretación específica en la teoría de la computación, con base en la función de transición δ, porque en esta ocasión la función no le asocia a las parejas formadas por $Q \times Sigma$ un elemento del conjunto

$$\mathbf{R} = (Q \cup \{q_Y, q_N\}) \times \Sigma \times \{\rightarrow, \leftarrow, -\}$$

sino con un subconjunto $R \subseteq \wp(\mathbf{R})$, donde $\wp(\mathbf{R})$ denota al conjunto potencia de $\mathbf{R}$. Lo que significa que no se podrá especificar cual de los elementos de R es la tripleta indicada para encontrar un estado de paro, después de un número finito de iteraciones.

Definición 6 *Una* máquina de Turing no determinística *bf (MTND) es una máquina de Turing con una función de transición no determinística definida como:*

$$\delta : Q \times \Sigma \rightarrow R \subseteq \wp((Q \cup \{q_Y, q_N\}) \times \Sigma \times \{\rightarrow, \leftarrow, -\})$$

Este concepto considera el caso de la máquinas deterministas, porque cualquier tripleta definida por la función de transición determinística está contenida en $\wp(\mathbf{R})$.

Lo anterior hace pensar que la idea de no determinísmo es más general que la idea de determinísmo, porque todo lo que se pueda representar en un modelo determinístico siempre se podrá representar en uno no determinístico, esto sería siempre que existíera un lenguajes admitido por una **MTND** que no fuera admintido por una **MTD**. Sin embargo, se puede demostrar que la clase de lenguajes que son aceptados por una máquina determinística es equivalente a la clase de lenguajes reconocidos por una no determinística. (ver *[Brookshear (1989)]*)

El funcionamiento usual de una máquina no determinística es semejante al de una máquina determinística, para toda cadena x, con la salvedad de que en cada paso se escoge de R una tripleta de manera arbitraria (en algunos casos se pueden verificar todas las tripletas que estén en R, para cada pareja (q, σ) que intervenga), y si de esta manera se encuentra un estado de paro q_Y, entonces se dirá que la máquina no determinística N acepta a la cadena x, de lo contrario se dira que la máquina N rechaza a la cadena x.

Para fijar ideas, necesitamos una notación que nos permita analizar cómo la máquina ejecuta las instrucciones de la función de transiciones paso a paso.

Definición 7 *Sea* $M = (Q, \Sigma, \delta, q_0)$ *una máquina de Turing y sean* $x, y \in \Sigma$, *entonces una* descripción instantánea (di) *es una cadena*

$$z_1 q z_2 \in \Sigma^* \cdot Q \cdot \Sigma^*$$

Donde $z_1 = w_1 x$ *y* $z_2 = y w_2$, *además si* $z_1 = \varepsilon$ *entonces* $w_1 = \varepsilon$ *y* $x = b$. *Si* $z_2 = \varepsilon$ *entonces* $w_2 = \varepsilon$ *y* $y = b$.

El paso de la actual **di** a la siguiente **di**, representada por $s_1 q' s_2$, se denota como:

$$z_1 q z_2 \vdash s_1 q' s_2$$

debiendóse cumplir una de las siguientes condiciones:

1. si $\delta(q, y) = (q', y', \rightarrow)$, entonces $w_1 x q y w_2 \vdash w_1 x y' q' w_2$;
2. si $\delta(q, y) = (q', y', \leftarrow)$, entonces $w_1 x q y w_2 \vdash w_1 q' x y' w_2$;
3. si $\delta(q, y) = (q', y', -)$, entonces $w_1 x q y w_2 \vdash w_1 x q' y' w_2$.

Con base en la notación anterior denotaremos al lenguaje aceptado por una **MT** como:

Definición 8 *Sea* $M = (Q, \Sigma, \delta, q_0)$ *una máquina de Turing, el lenguaje aceptado por* M, *denotado* L_M, *lo representa el siguiente conjunto:*

$$L_M = \left\{ z \in \Sigma^* \;\middle|\; \exists z_1, z_2 \in \Sigma^* \text{ tal que } q_0 z \vdash \ldots \vdash z_1 q_Y z_2 \right\}$$

Es decir, la máquina M *aceptará a la cadena* z *siempre y cuando después de un cierto número de pasos se alcanza el estado de aceptación.*

Consideremos ahora, el ejemplo de una máquina de Turing que sume dos números enteros positivos.

Ejemplo 1 *Sea* $\mathbf{Z}^+$ *el conjunto de los números enteros positivos, y sean*

$$x, y \in \mathbf{Z}^+$$

Pregunta *¿Existe una* **MT** *que sea capaz de calcular* $x + y$*?*

Para contestar la pregunta, primero debemos tener una representación conveniente del conjunto $\mathbf{Z}^+$.

El esquema de codificación que utilizaremos en esta ocasión es la representación unaria para los enteros, es decir, consideraremos al alfabeto $\Sigma = \{1\}$, *entonces* $\{1\}^*$ *es el conjunto de cadenas finitas formadas por* 1*'s o* ε.

Sea la función: $w : \mathbf{Z} \rightarrow \{1\}^*$, $x \mapsto w(x)$

tal que

$$| w(x) | = x.$$

Supongamos ahora, que la concatenación de las cadenas $w(x)$ y $w(y)$ está separada sólo por el símbolo 0, es decir, las cadenas de entrada para la máquina M son del tipo $w(x)0w(y)$. Después de terminar con los cálculos se desea que en la cinta de M este la cadena $w(x+y)$, y que el estado sea q_Y.

En otras palabras, usando descripciones instantáneas, tenemos que encontrar un programa para la máquina M que sea capaz de llevarnos en una secuencia finita de pasos de la cadena $q_0w(x)0w(y)$ a la cadena $q_Yw(x+y)0$.

Por la manera en que están representados los números en la máquina M es fácil ver que su suma es simplemente la concatenación de sus codificaciones. La máquina que se propone es la siguiente:

Sea $M = (Q = \{q_0, q_1, q_2, q_3, q_Y\}, \Sigma = \{1,0\}, \delta, q_0)$ donde δ se define en seguida:

$$\begin{aligned}
1: &\ \delta(q_0,1) = (q_0,1,\rightarrow)\\
2: &\ \delta(q_0,0) = (q_1,1,\rightarrow)\\
3: &\ \delta(q_1,1) = (q_1,1,\rightarrow)\\
4: &\ \delta(q_1,b) = (q_2,b,\leftarrow)\\
5: &\ \delta(q_2,1) = (q_3,0,\leftarrow)\\
6: &\ \delta(q_3,1) = (q_3,1,\leftarrow)\\
7: &\ \delta(q_3,b) = (q_Y,1,\rightarrow)
\end{aligned}$$

Para explicar el funcionamiento de la máquina supóngase la siguiente cadena de entrada (instancia):

Sea $x = 4$ y $y = 3$, entonces $w(x) = 1111$ y $w(y) = 111$. Es claro que la cadena que se quiere encontrar es $w(x+y) = 1111111$. Usando las **di's** *tenemos que:*

$$\begin{aligned}
&..bq_01111011b.. \vdash_1 ..b1q_01110111b.. \vdash_1 ..b11q_0110111b.. \vdash_1\\
&..b111q_010111b.. \vdash_1 ..b1111q_00111b.. \vdash_2 ..b11111q_1111b.. \vdash_3\\
&..b111111q_111b.. \vdash_3 ..b1111111q_11b.. \vdash_3 ..b11111111q_1b.. \vdash_4\\
&..b111111q_211b.. \vdash_5 ..b111111q_3101b.. \vdash_6 ..b11111q_31101b.. \vdash_6\\
&..b1111q_311101b.. \vdash_6 ..b111q_3111101b.. \vdash_6 ..b11q_31111101b.. \vdash_6\\
&..b1q_311111101b.. \vdash_6 ..bq_3111111101b.. \vdash_6\\
&..bq_3b11111101b.. \vdash_6 ..bq_Y11111101b.. \vdash_6
\end{aligned}$$

Donde $\vdash_i$ denota el i-ésimo paso del algoritmo definido por la función de transición de la máquina.

Es importante observar que en este caso la máquina de Turing está transformando una cadena de entrada, que representa una instancia de un problema, en una cadena de salida, la cual representa la solución de la instancia.

En las siguientes secciones usaremos a las máquinas de Turing para definir las clases de problemas: **P**, **NP** y **NP-C**, al igual que el concepto de *algoritmo eficiente*.

3.2 Modelos Determinísticos y la Clase P

En la parte anterior se habló de las propiedades que deben tener las máquinas de Turing en general y se dieron algunas ideas de lo que son, sin dirigirlas hacia un punto en particular; lo que se pretende hacer aquí es mostrar un modelo equivalente de las máquinas de Turing, al que vimos anteriormente, que nos permita construir las clases de problemas más importantes.

El modelo utilizado será el propuesto por *[Garey, Johnson (1979)]*, el cual se refiere a la Máquina de Turing Determinística de una Cinta (*deterministic one-tape Turing machine*), y se denotará **MTD**, la cual está representada esquemáticamente en la *figura 3-1*.

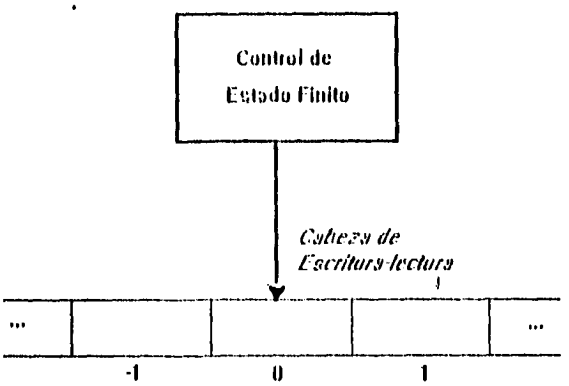


figura 3-1 Máquina de Turing Determinística

La **MTD** consiste de un *Control de estado finito*, una *Cabeza de escritura-lectura* y una *Cinta* formada por una sucesión infinita de celdas en ambos

sentidos, etiquetadas con ...-3,-2,-1,0,1,2,3....

Dado un alfabeto Σ un programa para una **MTD** especifica la siguiente información:

1. Un conjunto finito $\overline{\Sigma}$ de símbolos de la cinta, incluyendo un subconjunto $\Sigma \subset \overline{\Sigma}$ de símbolos de entrada y una distinción de símbolos blancos $b \in \overline{\Sigma} - \Sigma$

2. Un conjunto finito de estados Q que incluye un estado inicial q_0 y dos *estados de paro* q_Y y q_N.

3. Una función de transición

$$\delta : ((Q - \{q_Y, q_N\}) \times \overline{\Sigma}) \rightarrow (Q \times \overline{\Sigma} \times \{\rightarrow, \leftarrow\})$$

La operación de un programa se da como sigue: la entrada de una **MTD** es una cadena $x \in \Sigma^*$, la cual es colocada en la cinta, un símbolo por celda, desde la celda 1 hasta la celda $|x|$, y todas las demás celdas son inicializadas con el símbolo blanco, el programa empieza sus operaciones en el estado inicial q_0, con la cabeza de escritura-lectura en la celda 1. Y tenemos que en el $i-ésima$ paso la función de transición puede estar definidad como: $\delta(q,s) = (q', s', D)$ lo cual se interpreta de la siguiente manera:

La cabeza de lectura-escritura borra el símbolo s, escribe el símbolo s' en la misma celda de s se mueve una celda a la derecha, si $D = \rightarrow$, o una celda a la izquierda, si $D = \leftarrow$. En ese momento el control de estado finito cambio del estado q al estado q', y se actualizo nuevamente la función de transición a $\delta(q', s') = (p, t, D)$ este paso se repite hasta que el estado actual q es q_Y ó q_N, los cálculos terminan con la respuesta *sí*, si $q = q_Y$ o *no*, si $q = q_N$.

En otro caso el estado actual q pertenece a $Q - \{q_Y, q_N\}$ y algún símbolo $s \in \overline{\Sigma}$ que esté en la cinta es considerado, y el valor $\delta(q,s)$ se vuelve a definir.

Se dirá que un programa M para una **MTD** con un alfabeto de entrada Σ acepta a x si y sólo si M se detiene en un estado q_Y cuando x es la entrada.

Además el lenguaje L_M reconocido por el programa M está dado por:

$$L_M = \{x \in \Sigma^* \mid M \text{ acepta a } x\}$$

Observación: Si $x \in (\Sigma^* - L_M)$, entonces los cálculos en x pueden parar en el estado q_N o pueden continuar para siempre.

Definición 9 *Un* algoritmo *es una máquina de Turing la cual tiene que parar en todas las posibles cadenas sobre su alfabeto de entrada.*

También se dirá que el programa M de la **MTD** resuelve el problema de decisión Π bajo el esquema de codificación **e**, si M se detiene para todas las cadenas de entrada sobre su alfabeto y $L_M = L[\Pi, \mathbf{e}]$. Es decir, el lenguaje que representa al problema Π bajo el esquema de codificación **e** ($L_M = L[\Pi, \mathbf{e}]$, ver *cap II*) es el lenguaje que debe ser aceptado por la máquina M.

El *tiempo usado* en los cálculos de un programa M de una **MTD** con una entrada $x \in \Sigma^*$ es el número de pasos ocurridos hasta que se introduce algún estado de paro (q_Y o q_N).

Para un programa M de una **MTD** que termine para una entrada $x \in \Sigma^*$, la *función de trabajo* (la función que mide la complejidad del programa con respecto al tiempo) $\Upsilon_M : \mathbf{Z} \rightarrow \mathbf{Z}$ está dada por:

$$\Upsilon_M(n) = \left\{ m \;\middle|\; \begin{array}{l} \exists x \in \Sigma^*, |x| = n \text{ tal que los calculos en } M \\ \text{para la entrada } x \text{ toman tiempo } m \end{array} \right\}$$

donde **Z** es el conjunto de los números enteros.

El programa M es llamado programa para una **MTD** de *tiempo polinomial* si *existe un polinomio* p tal que, $\Upsilon_M(n) \leq p(n)$.

De lo anterior se desprende la formalización de la clase de lenguajes **P**, la cual se define como:

$$\mathbf{P} = \left\{ L \;\middle|\; \begin{array}{l} \exists M \text{ para una } \mathbf{MTD} \text{ de tiempo polinomial} \\ \text{para la cual } L = L_M \end{array} \right\}$$

Diremos que el lenguaje $L[\Pi, \mathbf{e}]$ que representa a un problema de decisión Π bajo el esquema de codificación **e** está en la clase **P** si hay un programa en tiempo polinomial para una **MTD** M que acepta al lenguaje, es decir, para toda cadena x en $L[\Pi, \mathbf{e}]$ la máquina M debe detenerse en el estado q_Y.

Un aspecto importante de aclarar esque cuando tengamos el lenguaje

$$L[\Pi, \mathbf{e}] \subset \mathbf{P},$$

abusando de la notación escribiremos $\Pi \in \mathbf{P}$ y diremos que el problema Π está en la clase de problemas polinomiales, aunque realmente quién está es $L[\Pi, \mathbf{e}]$.

3.3 Modelos no Determinísticos y la Clase NP

En esta sección se estudiará el modelo de computación correspondiente a la máquina no determinística de Turing.

Para iniciar con la presentación, pensemos en una instancia para el problema del agente viajero (**pav**) mencionado en la *sección 2.2* y supongamos que adivinamos (de manera completamente arbitraria) una posible sucesión de ciudades, y como sabemos el valor de las distancias entre cada par de ciudades, bastará con sumar aquellas que estén en el paseo propuesto. Esto se puede hacer con un algoritmo simple con complejidad en tiempo de $\mathbf{O}(p(n))$, donde n es el número de ciudades.

El hecho que se pueda adivinar, de alguna forma, una posible solución para algún problema y luego a esta solución poderla verificar en tiempo polinomial hace a un procedimiento no determinístico.

Informalmente se puede definir a la clase de problemas **NP** con el concepto que se llamará *algoritmo no determinístico*. Un algoritmo de esta clase está compuesto de dos fases.

- *Fase no-determinística o adivinadora*: Alguna cadena x completamente arbitraria de símbolos se escribe en la cinta.
- *Fase determinística o verificadora*: Esta fase funciona como una **MTD** usual.

El número de pasos ocupados durante el tiempo de ejecución de un algoritmo no determinístico es la suma de los pasos requeridos en las dos fases; es decir, el número de pasos que toma escribir la cadena x más el número de pasos ejecutados por la fase determinística.

Con esto, la clase **NP** se puede definir como la clase de todos los problemas de decisión que pueden ser resueltos por algoritmos no determinísticos en *tiempo polinomial*. La formalización de lo anterior es posible con la definición de lenguaje y *máquinas de Turing no determinísticas* de una cinta denotada **MTND**.

El modelo **MTND** que se describirá tiene la misma estructura que la **MTD**, excepto que esta tiene un módulo adivinador que tiene su propia cabeza de escritura, como se ve en la *figura 3-2*. Esta cabeza suministra el significado del valor adivinado para ser escrito en la cinta, y el módulo adivinador será usado solamente para este propósito.

Un *programa N* para una **MTND** se específica de la misma manera que en una **MTD**, incluyendo el alfabeto $\overline{\Sigma}$ de la cinta, el alfabeto de entrada Σ, el símbolo blanco b, el conjunto de estados Q, el estado inicial q_0, los estados de paro q_Y y q_N, y la función de transición definida como:

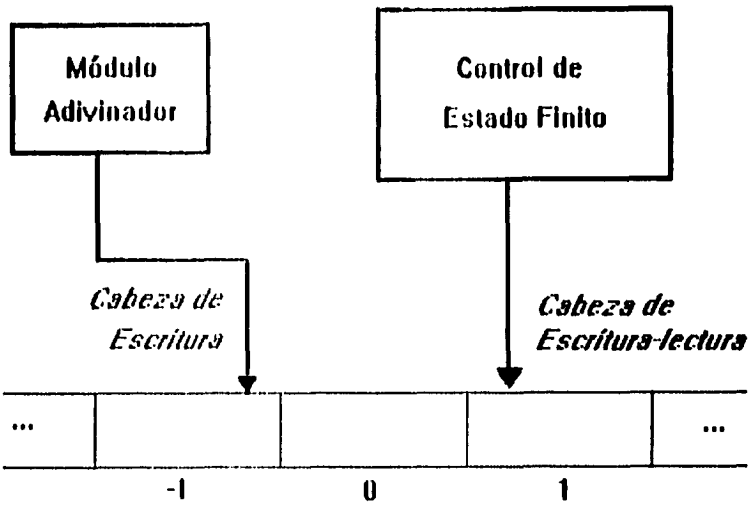


figura 3-2 Máquina de Turing no Determinística

$$\delta : Q - \{q_Y, q_N\} \times \overline{\Sigma} \rightarrow \wp(Q \times \overline{\Sigma} \times \{\rightarrow, \leftarrow\})$$

El cáculo de una cadena de entrada $x \in \Sigma^*$ en un programa para una **MTND** difiere de una **MTD** en que se realiza en dos fases.

La primera fase es *adivinadora*. Inicialmente la cadena de entrada x se escribe en la cinta de la celdas 1 a $|x|$ (mientras todas las demás celdas son inicializadas con el símbolo b), la cabeza de *escritura-lectura* está explorando la celda 1, la cabeza escritora está posicionada en la celda -1, y el control de estado finito está *desactivado*. Entonces el módulo adivinador dirige, en un paso tanto a la *cabeza de escritura* para que también escriba algún símbolo de Σ en la celda de la cinta, que está siendo explorada, y se mueve una celda a la izquierda, o para, con lo que el módulo adivinador se desactiva, mientras el control de estado finito se activa en el estado q_0.

Pero el módulo adivinador puede seguir activado, y de ser así algún símbolo de Σ puede ser escrito en la cinta, esto se hace de manera arbitraria por el módulo adivinador. De esta manera el módulo adivinador puede escribir cualquier cadena $x \in \Sigma^*$ antes de terminar o nunca parar.

La *fase verificadora* empieza cuando el control de estado finito es activado en el estado q_0; a partir de aquí los cálculos se realizan bajo la dirección de un programa **MTND** de acuerdo con una **MTD**.

El módulo adivinador y su cabeza escritora no son involucrados más, ha-

biendo terminado su labor de escribir la cadena adivinada en la cinta. La cadena adivinada puede ser examinada durante la fase verificadora y los cálculos terminan si el control de estado finito introduce uno de los dos estados de paro (q_Y, q_N). Entonces se dice que acepta un cálculo si el estado de paro es q_Y. Los demás cálculos pueden parar o no y son considerados como cálculos no aceptados.

Con lo que respecta al *tiempo requerido* por un programa N, para una **MTND** que acepta una cadena $x \in L$, está definido como el máximo sobre todos los cálculos aceptados por N para la entrada x y el número de pasos ocurridos en las fases adivinadora y verificadora hasta que se introduce el estado de paro q_Y.

La función de trabajo $\Upsilon_N : Z \to Z$ está dada como:

$$\Upsilon_N(n) = \max\left\{\{1\} \cup \left\{m \;\middle|\; \begin{array}{l}\exists x \in L, |x| = n \text{ tal que el tiempo} \\ \text{que ocupa } N \text{ para aceptar } x \text{ es } m\end{array}\right\}\right\}$$

Note que la función de trabajo para N depende solamente del número de pasos ocurridos en cálculos *aceptados;* y por convención, $\Upsilon(n) = 1$ cada vez que las entradas de longitud n no son aceptadas por N.

El programa N de la **MTND** es un programa de tiempo polinomial si existe un polinomio p tal que $\Upsilon(n) \leq p(n)$ para todo n.

Por último la clase **NP** estará formalmente definida como:

$$\mathbf{NP} = \left\{L \;\middle|\; \begin{array}{l}\exists N \text{ para una } \mathbf{MTND} \text{ de tiempo polinomial} \\ \text{para la cual } L = L_N\end{array}\right\}$$

De nueva cuenta es importante aclarar que si el leguaje $L[\Pi, \mathbf{e}]$ está contenido en **NP**, diremos que es el problema es **NP**.

Otra definición alternativa de esta clase de problemas, propuesta por *[Papadimitriou, Steiglitz (1981)].*

Definición 10 *Sea* **e** *un esquema de codificación sobre el alfabeto* Σ, *para el problema* Π, $\Pi \in$ **NP** *si existe un polinomio* $p(n)$ *y una máquina de Turing* N *tal que se cumple lo siguiente:*

La cadena $x \in Y_\Pi$ *si y sólo si existe una cadena de símbolos,* $c(x)$ *de* Σ^*, *tal que* $|c(x)| \leq p(|x|)$ *con la propiedad de que si en* N *se introduce la cadena* $xbc(x)$, *entonces se encuentra el estado de paro* q_Y *después de a lo más* $p(|x|)$ *pasos.*

3.4 Relación entre P y NP

La primera observación es que $\mathbf{P} \subset \mathbf{NP}$ y se debe a que cualquier problema de decisión que pueda ser resuelto por un algoritmo determinístico en tiempo polinomial también podrá ser resuelto por un algoritmo no determinístico en tiempo polinomial. Es decir, si el leguaje puede ser aceptado por una **MTD**, necesariamente será aceptado por una **MTND**, por como están construidas sus funciones de transición.

> **Observación:** Si un problema $\Pi \in \mathbf{P}$ y A es un algoritmo de tiempo polinomial para una **MTD** que resuelve a Π podemos construir un algoritmo no determinístico que resuelve Π simplemente usando A como la fase de verificación e ignorando la fase adivinadora. Luego entonces se tiene que:
>
> $$\Pi \in \mathbf{P} \Rightarrow \Pi \in \mathbf{NP}\square$$

Sin embargo, el punto interesante es: ¿$\mathbf{P} = \mathbf{NP}$ o $\mathbf{P} \subset \mathbf{NP}$?, esto es, ¿la noción de no determinísmo es más poderosa que la de determinísmo? Es decir, el sentido de que algunos problemas pueden ser resueltos en tiempo polinomial con un algoritmo no determinístico pero no se sabe si pueden ser resueltos en tiempo polinomial por un algoritmo determinístico.

En esta parte debemos notar que no es suficiente que la clase de lenguajes que aceptan las **MTD** sea la misma que la que aceptan las para aseverar que su capacidad para resolver problemas es la misma. Esto se debe a que dentro de la misma clase de lenguajes podemos hacer una distinción simple, pues si tenemos un problema Π representado por un cierto lenguaje $L[\Pi, \mathbf{e}]$ bajo un esquema de codificación $\mathbf{e}$ (supongamos que $\mathbf{e}$ es el mejor esquema que representa a Π) y además $L[\Pi, \mathbf{e}]$ es aceptado por una **MTD**, entonces $L[\Pi, \mathbf{e}] \in \mathbf{P}$, lo que significaría que Π puede ser resuelto por un algortimo en tiempo polinomial. Sin embargo, si $L[\Pi, \mathbf{e}]$ es aceptado en tiempo polinomial por una **MTND**, para mostrar que $L[\Pi, \mathbf{e}] \in \mathbf{P}$ tendríamos que simular la ejecución de la **MTND** con una **MTD**, lo que en general es más costoso con respecto al tiempo de ejecución que la misma **MTND**, pue como veremos más adelante cuando un problema Π es resuelto por una **MTND** bajo un esquema de codificación $\mathbf{e}$, entances existirá un algortimo de orden esponencial que lo resuelva.

Si algún problema Π está contenido en la clase **NP** y se puede resolver en tiempo polinomial, se puede dar (en forma determinística) una respuesta (si o no) si se verifican todas las cadenas de a lo más $p(n)$, es decir, correr la segunda

fase de un algoritmo no determinístico para cada cadena a la vez. Pero si cada cadena contiene c símbolos, entonces hay $c^{p(n)}$ cadenas de longitud $p(n)$, donde el número de cadenas es exponencial y no polinomial con respecto a n. De donde se tiene el siguiente resultado.

Teorema 1 *Si* $\Pi \in$ **NP**, *entonces existe un polinomio* p *tal que* Π *puede ser resuelto por un un algoritmo determinístico con complejidad en tiempo* $\mathbf{O}(2^{p(n)})$.

Demostración: *La prueba se obtiene con lo que se argumentó anteriormente, considerando que si existe un algoritmo* A *no determinístico de tiempo polinomial que resuelve* Π, *entonces el tiempo de ejecución de* A *está dado por un polinomio* $q(n)$. *Además podemos construir un algoritmo no determinístico* A' *tal que pruebe cada una de las* c *posibilidades hasta que encuentre una cadena cuya respuesta sea si.*

Por lo tanto, el tiempo de ejecución de A' *es*

$$q(n) \cdot c^{p(n)} \in \mathbf{O}(2^{p(n)})$$

para algún $p(n)$ *adecuado* □

Se cree que la clase **NP** es más grande que la clase **P**, pues no hay un sólo problema $\Pi \in$ **NP** para el cual se haya probado que esté en **P**, es decir, no existe, hasta ahora, un algoritmo determinístico de tiempo polinomial que resuelva a Π.

Sin embargo, la conjetura de que **NP** $\neq$ **P** es más razonable; si consideramos que si **P** = **NP**, significaría que existe un algoritmo en tiempo polinomial para cada problema que estuviera en la clase **NP**.

Pero como dicho algoritmo no ha sido encontrado se tiene la conjetura **NP** $\neq$ **P**.

Así, para todas las proposiciones o propiedades que tengan que usar la relación entre los problemas **NP** y **P** se supondrá que **NP** $\neq$ **P**.

3.5 Clase de Problemas NP-Completos

Hablar de problemas **NP-Completos (NP-C)** es hablar de los posibles problemas contenidos en **NP** − **P**, los cuales tienen la propiedad que si alguno de ellos tiene un algoritmo determinístico de tiempo polinomial que lo resuelva, entonces todos los problemas en **NP-C** podrían ser resueltos en tiempo polinomial por un algoritmo determinístico.

Para formalizar estas ideas necesitamos considerar lo siguiente. Supongamos que se quiere resolver un problema Π y se tiene además un algoritmo para

un problema Π', supongamos que tenemos una función que a cada instancia I del conjunto de instancia D_Π le asocia una instancia I' del conjunto de instancias $D_{\Pi'}$, especificada de la siguiente manera:

$$T : D_\Pi \rightarrow D_{\Pi'}$$

tal que $\forall x \in D_\Pi$ la respuesta correcta de Π en x es *sí* (es claro que $x \in Y_\Pi$) si y solamente si la respuesta correcta de Π' con $T(x)$ es *sí* es decir $T(x) \in Y_{\Pi'}$.

De esta manera con el algoritmo de Π' y la composición T se tiene un algoritmo para Π. Para fijar ideas consideremos la siguiente definición.

Definición 11 *Sea la función T como se definió anteriormente. T es una reducción polinomial (transformación polinomial) de Π a Π' si:*

1. *T puede ser acotada en tiempo polinomial, es decir el tiempo que ocupa la transformación es menor o igual a algún polinomio y se denotará como*

$$T(x) \leq p(n), |x| = n$$

2. *$\forall x \in D_\Pi$ la respuesta correcta en Π con x es la misma para Π' bajo $T(x)$.*

Definición 12 *Sean Π_1 y Π_2 dos problemas de decisión. Π_1 es* reducible polinomialmente *a Π_2 si existe una transformación polinomial de Π_1 a Π_2 y se escribe $\Pi_1 \propto \Pi_2$.*

La *definición 12* indica que Π_1 es *al menos tan difícil de resolver* como Π_2.

Si Π_2 tuviera un algoritmo en tiempo polinomial que lo resolviera significaría que Π_1 sería también soluble en tiempo polinomial, por lo que se considera el siguiente teorema.

Teorema 2 *Sean Π_1 y Π_2 dos problemas de decisión, si $\Pi_1 \propto \Pi_2$ y $\Pi_2 \in \mathbf{P}$, entonces $\Pi_1 \in \mathbf{P}$.*

Demostración: *Como $\Pi_1 \propto \Pi_2$, entonces existe $T : D_{\Pi_1} \rightarrow D_{\Pi_2}$ y por definición se cumple que*

$$T(x) \leq p(n), para x \in D_{\Pi_1} donde \mid x \mid = n$$

para algún polinomio p.

Además existe un polinomio q que acota a Π_2 y como x es la respuesta correcta para Π_1 si y sólo si $T(x)$ lo es para Π_2, tenemos por las propiedades de los polinomios que $q(T(x)) \leq q(p(n))$. De esta manera el costo total del algoritmo que resuelve Π_1 bajo T es

$$p(n) + q(p(n)) \ \square$$

Otro resultado que será de utlidad para seguir demostrando más propiedades de las reducciones polinomiales es:

Definición 13 *Sean* Π_1 *y* Π_2 *problemas de decisión, si* $\Pi_1 \propto \Pi_2$ *y* $\Pi_2 \propto \Pi_1$, *entonces se dice que son problemas* polinomialmente equivalentes.

Dentro de las propiedades que destacan en las reducciones polinomiales está aquélla que considera a $\propto$ como una *relación de orden parcial*, lo anterior se verifica con el próximo teorema.

Teorema 3 $(\Psi, \propto)$ *induce una relación de orden parcial, donde* Ψ *es el conjunto de los problemas de decisión.*

Demostración: *Se debe probar que* $\propto$ *es reflexiva, antisimétrica y transitiva. Supongamos que* $\Pi_i \in \Psi; i \in \{0,1,2,3\}$ *son escogidos arbitrariamente.*

(a) Reflexividad: $\Pi_0 \propto \Pi_0$.
Se debe mostrar que existe $T : D_{\Pi_0} \to D_{\Pi_0}$ *tal que*

$$\forall x \in D_{\Pi_0}, x \in Y_{\Pi_0} \text{ si y sólo si } T(x) \in Y_{\Pi_0}$$

es decir, x *es la respuesta correcta para* Π_0 *si y solo si* $T(x)$ *lo es también para* Π_0, *entonces sea* T *la función identica*

$$I : D_{\Pi_0} \to D_{\Pi_0}$$

donde

$$I(x) = x$$

(b) Antisimetría: Sean Π_1 *y* Π_2 *dos problemas de decisión, si* $\Pi_1 \propto \Pi_2$ *y* $\Pi_2 \propto \Pi_1$.
Por la polinomialmente equivalentes, lo que significa que ambos tienen el mismo grado de dificultad.

(c) Transitividad: Sean Π_1, Π_2, Π_3 *problemas de decisión;*

$$\text{si } \Pi_1 \propto \Pi_2 \text{ y } \Pi_2 \propto \Pi_3 \text{ entonces } \Pi_1 \propto \Pi_3$$

Por hipótesis sabemos que como $\Pi_1 \propto \Pi_2$, *entonces*

$$\exists T_1 : D_{\Pi_1} \to D_{\Pi_2}$$

tal que $x \in Y_{\Pi_1}$ *si y solo si* $T_1(x) \in Y_{\Pi_2}$.
Análogamente, como $\Pi_2 \propto \Pi_3$ *se tiene que*

$$\exists T_2 : D_{\Pi_2} \to D_{\Pi_3}$$

tal que $y \in Y_{\Pi_2}$ *si y solo si* $T_2(y) \in Y_{\Pi_3}$

Por demostrar que

$$\exists T : D_{\Pi_1} \rightarrow D_{\Pi_3}$$

tal que $\forall x \in Y_{\Pi_1}$ *si y sólo si* $T(x) \in Y_{\Pi_3}$.

Como $x \in Y_{\Pi_1}$ *si y solo si*

$$T_1(x) \in Y_{\Pi_2}$$

y además sabemos que

$$T_1(x) \in Y_{\Pi_2}$$

si y sólo si

$$T_2(T_1(x)) \in Y_{\Pi_3}$$

Entonces sea T *la composición* $T_2 \circ T_1$ □

Ahora ya estamos preparados para dar una definición formal de los problemas **NP-C**, la cual es:

Definición 14 *Sea* Π *un problema de decisión, el leguaje* $L[\Pi, \mathbf{e}]$ *que representa a* Π *bajo el esquema de codificación* **e** *está en la clase de problemas* **NP-C**, *si* $L[\Pi, \mathbf{e}] \in \mathbf{NP}$ *y además* $\forall L[\Pi', \mathbf{e}] \in \mathbf{NP}$, $\Pi' \propto \Pi$.

De la misma manera, como en los casos anteriores para las clases de problemas **P** y **NP**, diremos que cualquier problema de decisión Π que cumpla con la definción anterior está en la clase **NP-C**, en lugar de decir que es $L[\Pi, \mathbf{e}]$ quien efectivamente pertenece a esa clase.

Es importante notar que el posible inconveniente de esta definición, es que dado un problema sería difícil demostrar la segunda parte de la definición, de hecho esta parte es considerada por algunos autores como la parte dura de un problema **NP-C**. *[Baase(1988)]*

Se ha vistó que una manera fácil de demostrar si un problema es **NP-C** es utilizando la propiedad de transitividad de la relación $\propto$ la cual se puede ver a partir del siguiente resultado:

Corolario 1 *Sean* Π *y* Π' *dos problemas de decisión. El problema* $\Pi \in \mathbf{NP} - \mathbf{C}$ *si:*

1. $\Pi \in \mathbf{NP}$
2. $\exists \Pi' \in \mathbf{NP\text{-}C}$ *tal* $\Pi' \propto \Pi$

Demostración: *Como* $\Pi_1 \in$**NP-C** *entonces para todo* $\Pi_i \in$**NP** *existe una transformación en tiempo polinomial tal que* $\Pi_i \propto \Pi_1$, *y por transitividad* $\Pi_i \propto \Pi$, *además como* $\Pi \in$**NP**, *entonces se cumple que* $\Pi \in$**NP-C** □

De esta manera, la situación que se nos presenta es encontrar un problema $\Pi \in$ **NP** – **C** del cual se pueda empezar, para demostrar que la clase **NP-C** no es vacía, y posteriormente explorar está clase.

3.6 Teorema de Cook

El primer problema **NP** – **C** fue un problema de decisión de lógica booleana llamado *Problema de Satisfacción de claúsulas* (*Satisfiability problem*) para abreviar **sat**. *[Cook(1971)]*. Para hacer un planteamiento formal de este problema necesitamos las siguientes definciones.

Definición 15 *Un alfabeto para lógica proposicional consta de:*

1. *Un conjunto contable de símbolos proposicionales,* $SP = \{P_1, P_2, ...P_n\}$
2. *Los conectivos lógicos usuales,* $\vee, \wedge, \Rightarrow, \neg$ *y el símbolo* $\perp$ *que tiene el valor de verdad de falso.*
3. *Símbolos auxiliares,* ")","(".

A continuación definiremos el conjunto de todas las fórmulas proposicionales (proposiciones) que pueden ser formadas por SP, usando el concepto de cerradura inductiva[1], del conjunto de la *definición 15*

Definición 16 *El conjunto PROP de formulas proposicionales es la cerradura inductiva del conjunto* $SP \cup \{\perp\}$ *bajo las funciones siguientes:*

1. $C_\neg(A) = \neg A$
2. $C_\vee(A, B) = A \vee B$

[1]Sea A un conjunto, $X \subset A, F = \{f \mid f : A^n \to A, n > 0\}$. Diremos que Y es *inductivo* en X si y sólo si $X \subset Y$ y Y es cerrado bajo las funciones de F, *i.e.* $\forall f \in F, \forall y_1, y_2, y_3, ..., y_n \in Y \Rightarrow f(y_1, y_2, y_3, ..., y_n) \in Y$. La intersección de todos los conjuntos inductivos en X son también cerrados bajo F y se llaman *cerradura inductiva de* X *bajo* F.

3. $C_{\wedge}(A,B) = A \wedge B$

4. $C_{\Rightarrow}(A,B) = A \Rightarrow B$

La definición 16 puede interpretarse recursivamente como sigue, ver [*Gallier(1986)*]:

> El conjunto $PROP$ es el conjunto más pequeño de cadenas sobre el alfabeto de la *definición 15* tal que:
>
> 1. Todo símbolo proposicional $P_i \in PROP$ y $\perp \in PROP$
> 2. Si $A \in PROP$, entonces $\neg A \in PROP$
> 3. Si $A, B \in PROP$, entonces $A \vee B \in PROP$, $A \wedge B \in PROP$, $A \Rightarrow B \in PROP$
> 4. $A \in PROP$ sólo si A está formada por las reglas anteriores.

Considerando la válidez sintáctica de las expresiones formadas por la definición anterior, lo siguiente que se debe tener son las definiciones de cuándo una *proposición está satisfecha* y la de cuando una proposición es *tautología* .

Definición 17 *Sea $B = \{T, F\}$ el conjunto de valores de verdad. Se supone, además, que B es un conjunto totalmente ordenado, $F < T$. (lo que más adelante nos permitirá asociar 0 con F y 1 con T).*

A continuación daremos la definición de los conectivos lógicos: $\vee, \wedge, \Rightarrow$ y $\neg$, que serán interpretados como una función H_* con $* \in \{\vee, \wedge, \Rightarrow\}$ definida como:

$$H_* : SP \times SP \rightarrow B,$$

y para la negación se tiene

$$H_{\neg} : SP \rightarrow B$$

Definición 18 *Las relaciones de los conectivos lógicos se representan con la siguiente tabla:*

P	Q	$H_{\neg}(P)$	$H_{\vee}(P,Q)$	$H_{\wedge}(P,Q)$	$H_{\Rightarrow}(P,Q)$
T	T	F	T	T	T
T	F	F	T	F	F
F	T	T	T	F	T
F	F	T	F	F	T

la constante lógica $\perp$ se interpreta como F.

Ahora definiremos la semántica de las fórmulas de $PROP$.

Definición 19 *Una* asignación de verdad *o* evaluación *es una función*

$$v : SP \to B$$

que asigna un valor de verdad a todos los símbolos proposicionales, y como $PROP$ *es* libremente generado[2] *por* SP, *ver [Gallier (1986)], todas las evaluaciones de* v *se extienden a una unica función*

$$\hat{v} : PROP \to B$$

tal que satisface las siguientes condiciones para todo $A, B \in PROP$:

(a) $\hat{v}(\bot) = F$
(b) $\hat{v}(P) = v(P), \forall P \in SP$
(c) $\hat{v}(\neg A) = H_{\neg}(\hat{v}(A))$
(d) $\hat{v}(A * B) = H_{*}(\hat{v}(A), \hat{v}(B)), * \in \{\vee, \wedge, \Rightarrow\}$

De esta manera podemos definir el concepto de tautología como sigue.

Definición 20 *Una proposición* A *es una tautología si y sólo si* $\hat{v}(A) = T$ *para toda evaluación* v,*y se abrevia como* $\models A$.

Diremos que una proposición se puede satisfacer si existe una evaluación v tal que $\hat{v}(A) = T$, de lo contrario se dirá que no se puede satisfacer.

También un concepto importante es el de *consecuencia lógica* a partir de un conjunto de proposiciones, el cual lo podremos plantear como:

$$\overline{X} \subseteq PROP$$

diremos que A es una consecuencia semántica (o consecuencia lógica) de $\overline{X}$, denotada como $\overline{X} \models A$, si para todas las evaluaciones $v, \hat{v}(A') = T$, para toda $A' \in \overline{X}$ implica que $\hat{v}(A) = T$.

[2]Sea A un conjunto, $X \subseteq A, F$ es un cojunto de funciones de A y X_+ es la cerradura inductiva de X bajo F. Se dice que X_+ es *libremente generado* para X y F si se cumple:

1. $f \restriction_{X_+^n} : A^m \to A \in F$ es inyectiva
2. $\forall f : A^m \to A, g : A^n \to A \in F, f(X_+^m)$ es disjunta de $g(X_+^n)$ siempre y cuando $f \neq g$
3. $\forall f : A^m \to A \in F$ y para todo $(x_1, x_2, x_3, ..., x_m) \in X_+^m$ se tiene que $f(x_1, x_2, x_3, ..., x_m) \in X$

El problema de determinar si una proposición se puede satisfacer es a lo que llamaremos **el problema del sat.** El tipo de proposiciones que se tomaran en cuenta para plantear el **sat**, serán aquellas que esten en *forma normal conjuntiva*, concepto que se definirá más adelante, pero antes necesitamos de las siguientes definiciones.

Definición 21 *Una* literal *es un símbolo proposicional o su negación. Dada una literal x su conjugada es $\overline{x}$ y se define como sigue:*

1. *Si $x = P$, entonces $\overline{x} = \neg P$*
2. *Si $x = \neg P$, entonces $\overline{x} = P$*

Además, una proposicion de la forma

$$C_i = B_{i,1} \vee B_{i,2} \vee B_{i,3} \vee \ldots \vee B_{i,n}$$

donde cada $B_{i,j}$ es una literal se llama claúsula.

Definición 22 *Una proposición A esta en* forma normal conjuntiva **(fnc)** *si es una conjunción de claúsulas.*

Restringir el problema del **sat** a proposiciones en **fnc** no debe causar ninguna inconsistencia pues para toda proposición $A' \in PROP$ existe una proposición A en **fnc** equivalente a A', es decir, podemos enunciar el siguiente teorema.

Teorema 4 *Para toda proposición $A' \in PROP$, existe una proposición A en* **fnc** *tal que $\models A' \equiv A$.*

Demostración. *La demostración se hará en forma constructiva, donde se dará un procedimiento general para transformar una fórmula $A' \in PROP$, en una fórmula en* **fnc**.

Paso 1. Usar los siguientes axiomas

$$P \Leftrightarrow Q = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$
$$P \Rightarrow Q = \neg P \vee Q$$

Paso 2. Usar repetidamente los siguientes axiomas

$$\neg(\neg P) = P$$
$$\neg(P \vee Q) = \neg P \wedge \neg Q$$

Paso 3. Usar repetidamente las leyes distributivas

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

Como una proposición de PROP es una cadena finita de símbolos proposicionales semántica y sintácticamente correctos, necesariamente en un número finito de pasos se debe encontrar una solución. Para ver con más detalle la demostración consultar [Gallier (1986)]□

Ejemplo 2 *Considere la siguiente proposición:*
$A' = (\neg P \Rightarrow Q) \Rightarrow (\neg R \Rightarrow S)$
por el paso 1 se tiene que:

$$A' = \neg(\neg P \Rightarrow Q) \vee (\neg R \Rightarrow S)$$
$$A' = \neg(\neg(\neg P) \vee Q) \vee (\neg(\neg R) \vee S)$$

por el paso 2 se tiene que:

$$A' = \neg(P \vee Q) \vee (R \vee S)$$
$$A' = (\neg P \wedge \neg Q) \vee (R \vee S)$$

por el paso 3 se tiene que:

$$A' = (\neg P \wedge \neg Q) \vee (R \vee S) = (R \vee S) \vee (\neg P \wedge \neg Q))$$
$$A' = ((R \vee S) \vee \neg P) \wedge ((R \vee S) \vee \neg Q)$$
$$A' = (R \vee S \vee \neg P) \wedge (R \vee S \vee \neg Q)$$

Donde es claro que A' está en **fnc.**

Dado lo anterior podemos plantear el **sat** como un problema de decisión de la siguiente manera:

Problema 3 *EL SAT* **(sat)**

Sea L conjunto de literales

$$L = \{x_1, x_2, x_3, ..., x_m\}$$

y sea

$$\Phi = \left\{ C_i \mid C_i = \bigvee_{x_i \in L} x_i \right\}$$

además con el conjunto Φ *formamos proposiciones de la forma*

$$A_\Phi = \bigwedge_{C_i \in \Phi} C_i$$

Pregunta: *¿Para toda propocisión* A_Φ *existe una evaluación* v *tal que* $\hat{v}(A_\Phi) = T$ *?*

Usualmente a v se le llama *asignación de verdad satisfactoria.*

Continuemos con el siguiente ejemplo:

Ejemplo 3 *Sea*

$$C = (x_1 \vee \overline{x}_2 \vee x_3)$$

$C = T$ *bajo* v, *es decir,* $\hat{v}(C) = T$

si y sólo si

$$v(x_1) = v(x_2) = v(x_3) = F$$

Si contamos el número de posibles asignaciones de verdad de C bajo v, tendríamos que es 2^3 para el caso anterior, pués cada literal puede tener dos valores de verdad y en el caso en que $| C | = r$ tenemos que el número posible de asignaciones es de 2^r.

Ejemplo 4 *Sea*

$$\Phi_1 = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2) \wedge (x_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3) \wedge (x_3 \vee \overline{x}_2)$$

Si

$$v(x_1) = T, v(x_2) = F, v(x_3) = F$$

entonces v *es una asignación de verdad satisfactoria.*

Ejemplo 5 *Si consideramos ahora la nueva proposición*

$$\Phi_1 \wedge \{\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3\}$$

veremos que no existe una asignación de verdad satisfactoria.

Podemos interpretar las proposiciones en **fnc** como expresiones de algebra booleana. Así, cuando tengamos una asignación verdadera la consideraremos como 1, y 0 en el otro caso. Además las literales tendrán las siguiente interpretación

$$x = x$$
$$\overline{x} = 1 - x$$

Como para que una proposición en **fnc** sea verdadera se tiene que cumplir

$$\hat{v}(A) = \hat{v}(C_1 \wedge C_2 \wedge C_3 \wedge ... \wedge C_k) = T$$

entonces

$$\hat{v}(C_i) = T, \forall i \in \{1..k\}$$

i.e. cada claúsula debe de ser verdadera, lo que implica que debe existir por lo menos para cada claúsula una literal que tenga un valor de verdad verdadero, por lo que para cada C_i se debe cumplir que

$$\sum_{x \in C_i} x + \sum_{\overline{x} \in C_i} (1 - x) \geq 1$$

Continuando con el ejemplo anterior podemos hacer el siguiente planteamiento:

$$\Phi_1 = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2) \wedge (x_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3) \wedge (x_3 \vee \overline{x}_2)$$

es equivalente a

$$\Phi_1' = (x_1 + x_2 + x_3) \cdot (x_1 + \overline{x}_2) \cdot (x_1 + \overline{x}_3) \cdot (x_2 + \overline{x}_3) \cdot (x_3 + \overline{x}_2)$$

pero $\hat{v}(\Phi_1) = T$ si y solamente si se cumplen las siguientes desigualdades para Φ_1':

$$x_1 + x_2 + x_3 \geq 1$$
$$x_1 + 1 - x_2 \geq 1$$
$$x_1 + 1 - x_3 \geq 1$$
$$x_2 + 1 - x_3 \geq 1$$
$$x_3 + 1 - x_2 \geq 1$$

también podemos transformar el conjunto anterior de ecuaciones como un *problema de programación lineal entera 0-1* (**pple01**) modificando la primera restricción para tenerla de la forma

$$x_1 + x_2 + x_3 \geq y$$

y agregando una función objetivo $z = y$ que trate de maximizar el valor de y, finalmente el problema queda como

$$\max z = y$$

$$x_1 + x_2 + x_3 \geq y$$
$$x_1 + 1 - x_2 \geq 1$$
$$x_1 + 1 - x_3 \geq 1$$
$$x_2 + 1 - x_3 \geq 1$$
$$x_3 + 1 - x_2 \geq 1$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

lo relevante de esta transformación, que además es de tiempo polinomial, es que muestra una reducción del **sat** a un **pple01**, lo que nos indica que el **pple01** es tan difícil de resolver como el **sat.**

De lo anterior se sigue que cualquier problema que sea planteado como un **pple01** y no se le puedan relajar sus restricciones será tan difícil de resolver como el **sat.**

A continuación demostraremos un resultado importante para la clase **NP-C.** Este teorema muestra que la clase de problemas **NP-C** es no vacía, la demostración que se propone esta basada en la que se encuentra en [*Papadimitriou, Stiglitz (1982)*].

Teorema 5 $\mathbf{sat} \in \mathbf{NP} - \mathbf{C}$

Demostración. *Se tienen que probar dos cosas: la primera es que* $\mathbf{sat} \in \mathbf{NP}$ *y la segunda es que para todo* $\Pi \in \mathbf{NP}$ *se cumple que* $\Pi \propto \mathbf{sat}$.

Para demostrar que el $\mathbf{sat} \in \mathbf{NP}$, *propondremos un algoritmo no determinístico que resuelva el problema en tiempo polinomial, especificado como sigue:*

1. **for** $i = 1$ **to** n **do** $x_i \leftarrow escoger(\{T, F\})$
2. **if** $A_\Phi \models T$ **then** *éxito*
3. **else** *falla*

Notemos que el algoritmo tiene una complejidad $\mathbf{O}(n)$, *con esto es suficiente para decir que el* $\mathbf{sat}$ *es* **NP**.

Para demostrar la segunda parte, tenemos que probar que para toda cadena x *existe una fórmula en* **fnc** $F(x)$ *(usando sólo el hecho de que* $\Pi \in \mathbf{NP}$*) tal que* $x \in L_N$, *para alguna* **MT** *si y sólo si* $F(x)$ *se puede satisfacer. Como* $\Pi \in \mathbf{NP}$, *entonces existe una* **MTND** N *que acepta a* x *en tiempo polinomial, es decir existe un polinomio* p *tal* $\Upsilon(n) \leq p(n)$, *donde* $\mid x \mid = n$

Para construir la fórmula F *consideraremos las siguientes literales:*

1. *Una literal* $x_{ij\sigma}$ *para todo* $0 \leq i, j \leq p(n)$ *y* $\sigma \in \Sigma$. *El significado para esta literal es: en el momento* i *la* j*-ésima posición de la cadena* x *contiene al símbolo* σ.

2. *Una literal y_{ijl} para todo $0 \leq i \leq p(n)$, $0 \leq j \leq p(n)+1$ y $1 \leq l \leq |N|$, donde $|N|$ es el número de instrucciones de N. Esta literal tiene el siguiente significado: en el momento i se busca en la j-ésima posición y se empieza a ejecutar la l-ésima instrucción. Si $j = 0$ ó $j = p(n)+1$, quiere decir que se ha recorrido toda la cadena, entonces se tiene que el cálculo es inútil.*

Lo que haremos en seguida es combinar las literales en una fórmula $F(x)$, tal que $F(x)$ se puede satisfacer si $x \in Y_\Pi$ es decir, la máquina N termina con el estado q_Y, para la cadena x. Si las literales que se especificaron tienen el significado indicado, $F(x)$ garantizará que la la máquina N empezará, escencialmente, con la cadena x en su parte más izquierda, que puede ser eventualmente aceptada por N. Construiremos la fórmula $F(x)$ como la conjunción de cuatro fórmulas de la siguiente manera

$$F(x) = U(x) \wedge S(x) \wedge W(x) \wedge E(x)$$

1. *El propósito de $U(x)$ es asegurar que cada vez que en el momento i, para $0 \leq i \leq p(n)$, cada posición de la cadena contiene un único símbolo, la cabeza busca una posición única dentro de los límites de la cadena, y el programa ejecuta una sola declaración*

$$U(x) = \left(\bigwedge_{\substack{0 \leq i,j \leq p(n) \\ \sigma \neq \sigma'}} (\overline{x}_{ij\sigma} \vee \overline{x}_{ij\sigma'}) \right) \wedge \left(\bigwedge_{\substack{0 \leq i \leq p(n) \\ j \neq j' \text{ o } i \neq i'}} (\overline{y}_{ijl} \vee \overline{y}_{ij'l'}) \right) \wedge$$
$$\wedge \left(\bigwedge_{\substack{0 \leq i \leq p(n) \\ 1 \leq l \leq |N|}} \overline{y}_{i0l} \wedge \overline{y}_{i,p(n)+1,l} \right) \wedge$$
$$\wedge \left(\bigwedge_{0 \leq i \leq p(n)} \left(\left(\bigwedge_{1 \leq j \leq p(n)} \bigvee_{\sigma \in \Sigma} x_{ij\sigma} \right) \wedge \bigvee_{\substack{1 \leq j \leq p(n) \\ 1 \leq l \leq |N|}} y_{ijl} \right) \right)$$

2. *Una fórmula $S(x)$ declara que la operación de N empieza correctamente, es decir, en el momento inicial los $n+1$ símbolos más a la izquierda de la cadena x se ven xb, la cabeza de escritura-lectura busca el símbolo más a la izquierda de la cadena, y las primeras instrucciones de N están a punto de ser ejecutadas*

$$S(x) = \left(\bigwedge_{j=1}^{n} x_{0j x(j)} \right) \wedge x_{0,n+1,b} \wedge y_{011}$$

donde la posición $x(j)$ es para el j-ésimo símbolo de la cadena x.

3. *La fórmula $W(x)$ afirma que N se desempeña bien, de acuerdo con las instrucciones del programa, $W(x)$ es la conjunción de las*

fórmulas $W_{ij\sigma l}$, una para cada $0 \leq i \leq p(n)$, $1 \leq j \leq p(n)$, $\sigma \in \Sigma$, $1 \leq l \leq |N|$, tal que la l-ésima instrucción de N es

$$l : \textbf{if } \sigma \textbf{ then } (\sigma'; 0; l')$$

La fórmual $W_{ij\sigma l}$ se define como:

$$W_{ij\sigma l} = \left(\overline{x}_{ij\sigma} \vee \overline{y}_{ijl} \vee x_{i+1,j,\sigma'}\right) \wedge \left(\overline{x}_{ij\sigma} \vee \overline{y}_{ijl} \vee y_{i+1,j+0,l'}\right) \wedge \bigwedge_{\tau \neq \sigma} \left(\left(\overline{x}_{ij\tau} \vee \overline{y}_{ijl} \vee x_{i+1,j,\tau}\right) \wedge \left(\overline{x}_{ij\tau} \vee \overline{y}_{ijl} \vee y_{i+1,j,l+1}\right)\right)$$

Esto significa que siempre que $x_{ij\sigma}$ y y_{ijl} son ambas verdaderas, en el siguiente momento, en el instante en que las variables x y y declaran que N realiza el movimiento correcto, estas deben ser ciertas. Para la última instrucción de N se tiene para cada i, j, σ

$$W_{ij\sigma|N|} = \left(\overline{x}_{ij\sigma} \vee \overline{y}_{ij|N|} \vee x_{i+1,j,\sigma}\right)$$

asegurando que una vez que el estado de aceptación es encontrado el proceso termina. Finalmente, $W(x)$ contiene las claúsulas

$$\bigwedge_{\substack{0 \leq i \leq p(n) \\ \sigma \in \Sigma \\ 0 \leq l \leq |N| \\ j \neq j'}} \left(\overline{x}_{ij\sigma} \vee \overline{y}_{ij'l} \vee x_{i+1,j,\sigma}\right)$$

lo que significa que siempre que N busca una posición diferente de la j-ésima, el símbolo de la j-ésima posición permanece sin cambios.

4. *La última parte de $F(x)$ garantiza que las operaciones de N terminan correctamente, esta parte consiste de una sóla claúsula*

$$E(x) = \bigvee_{j=1}^{p(n)} y_{p(n),j,|N|}$$

Esto completa la construcción de $F(x)$, primero observemos que esta construcción requiere sólo de una cantidad de tiempo polinomial con respecto al tamaño de la cadena x. Esto se tiene por el hecho de que de la longitud total (número de ocurrencias de las literales multiplicado por la longitud de los subíndices de estas literales en la fórmula F) es

$$\mathbf{O}(p^3(n) \cdot \log p(n))$$

Entonces para probar que la construcción es una transformación en tiempo polinomial de Π al **sat**, *se tiene que probar lo siguiente:*

- *$F(x)$ se puede satisfacer si y sólo si x es una instancia que está en Y_Π*

Para demostrar la parte sólo si, supongamos que $F(x)$ es satisfacible, entonces se satisfacen también

$$U(x), S(x), W(x) \text{ y } E(x)$$

con la misma asignación de verdad v. Como v satisface a $U(x)$, debe haber para todo i, j exactamente una variable $x_{ij\sigma}$ que sea verdadera, lo que significa que la j – ésima celda contiene a σ en el instante i. También para todo i una de las literales y_{ijl} es verdadera, lo que significa que en el momento i, la j-ésima posición de la cadena es explorada y la instrucción l es ejecutada por N. Finalmente ninguna variable de la forma

$$y_{i,p(n)+1,l} \text{ ó } y_{i0l}$$

puede ser verdadera, lo que se interpreta como que la cabeza nunca cae en la cadena. La asignación de verdad v, además, describe algunas secuencias de las cadenas, posiciones de la cabeza e instrucciones. Ahora demostraremos que esta secuencia es, de hecho, un cálculo válido, aceptado por N para una entrada $xbc(x)$, donde $c(x)$ es como en la definición 8.

Como $S(x)$ debe ser satisfecha por v, significa que la secuencia empieza correctamente, con los primeros $n+1$ lugares ocupados por la cadena x, y con el primer símbolo de x explorado mientras la primera instrucción es ejecutada.

El hecho de que $W(x)$ sea también satisfecha por v, significa que la secuencia cambia de acuerdo con las reglas de la función δ. Finalmente, v satisface a $E(x)$ sólo si N termina aceptando a x. Consecuentemente, si $F(x)$ es satisfacible, existe una cadena $c(x)$ de longitud finita menor que n tal que N acepta $xbc(x)$, por lo tanto $x \in Y_\Pi$.

Para demostrar la parte si, supongamos que $x \in Y_\Pi$. Entonces existe una cadena $c(x)$ de longitud $p(n)-n-1$ tal que N acepta $xbc(x)$. Esto quiere decir que existen una sucesión de $p(n)$ cadenas (cuyo elemento inicial es $xbc(x)$), los números de instrucciones y posiciones exploradas que son legales de acuerdo con N y que terminan con la aceptación de $xbc(x)$. Esta sucesión define una asignación de verdad v que necesariamente satisface a $F(x)$. Mostramos una transformación polinomial de Π al **sat**, *y como Π fue tomado arbitrariamente de* **NP**, *el teorema se ha demostrado.*□

La importancia de los problemas **NP-C**, además de su transfondo teórico, es que algunos de los problemas de optimización por ejemplo, el problema del agente viajero o el problema de clique, el problema de progrmación lineal entera, entre otros, pertenecen a esta clase. Una compilación detallada de la relación que guardan los problemas **NP-C** fue dada por *Garey, Jonhson (1979)*, en la cual destacan la relación de los problemas que se muestran en la *figura 3-3.*

En la figura el sentido de las flechas indica que existe una transformación de tiempo polinomial de un problema a otro, es decir existe una transformación de

tiempo polinomial, por ejemplo, del **sat** al **3-sat**. Se muestra el planteamiento de los problemas de la *figura 3-3* como problemas de decisión , y se dice además, para cada problema cómo son las transformaciones que los relacionan con los problemas adyacentes.

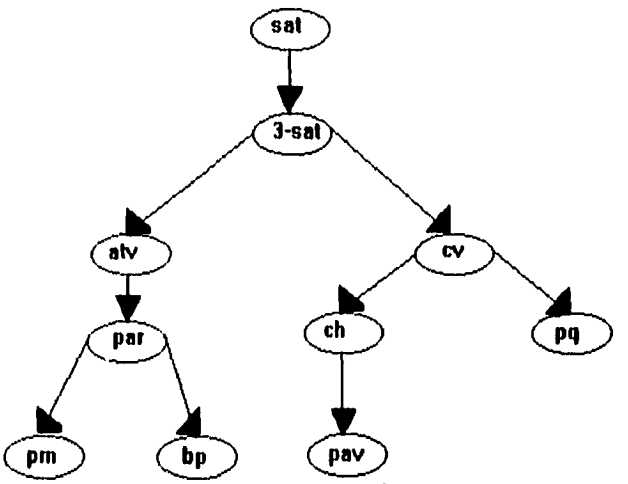


figura 3-3 Relación de los principales problemas NP-C

La manera en que explicaremos cada caso sera la siguiente: para demostrar que un problema, por ejemplo, el **3-SAT** está en **NP-C** se usará el resultado del *corolario 1*, es decir, se demuestra que el $\mathbf{3-SAT} \in \mathbf{NP}$ y mostramos explícitamente que existe T_{sat} en tiempo polinomial, tal que $\mathbf{sat} \propto \mathbf{3-sat}$ bajo T_{sat}. La proposición $\mathbf{3-SAT} \in \mathbf{NP-C}$ es realmente un teorema; sin embargo sólo mostraremos cómo es la transformación, para consultar la demostración detallada de las transformaciones de la *figura 3-3* ver *[Garey, Jhonson (1979)]* o *[Papadimitriou(1994)]*. Para los demás casos se puede consultar la transformación en las referencias anteriores.

Problema 4 3-SAT

Sea $C = \{c_1, c_2, c_3, ..., c_m\}$ *una colección de claúsulas de un conjunto finito de variables* U, *tales que* $| c_i | = 3$ *para* $1 \leq i \leq m$.

Pregunta: *¿Existe una signación de verdad para* U *que satisfaga todas las claúsulas en* C*?*

Transformación

La transformación T_{sat} en tiempo polinomial, tal que **sat** $\propto$ **3 − sat** *se construye de la siguiente manera:*

Sea L conjunto de literales

$$L = \{x_1, x_2, x_3, ..., x_m\}$$

y sea

$$\Phi_L = \left\{ C_i \mid C_i = \bigvee_{x_i \in L} x_i \right\}$$

además con el conjunto Φ formamos proposiciones de la forma

$$A_\Phi = \bigwedge_{C_i \in \Phi} C_i$$

que son instancias arbitrarias del **sat**.

Construiremos una proposición $A_{\Phi_{L'}}$, a partir de un conjunto de literales L', que especificaremos más adelante,

donde

$\Phi_{L'}$ es el conjunto de cláusulas con tres literales que se pueden formar con L', tal que $\hat{v}\Phi_{L'} = T$ si y sólo si $\hat{v}\Phi_{L'} = T$.

La construcción de $A_{\Phi_{L'}}$, se hará reemplazando cada cláusula C_j de $A\Phi_L$ por una conjunción de clade tres literales C'_j, las literales de estas cláusulas serán del conjunto L y algunas literales adicionales L'_j cuyo uso se limitará en la s cláusulas C'_j.

De esta manera la cláusula C' estará formada de la siguiente manera:

$$L' = L \cup \left\{ \bigcup_{j=0}^{m} L'_j \right\}$$

además

$$C' = \bigvee_{j=1}^{m} C'_j.$$

Para completar la transformación es necesario decir cómo tienen que ser construidos C'_j y L'_j, con respecto a una cláusula $C_j \in \Phi$.

Sea $C_j = z_1 \vee z_1 \vee z_2 \vee ... \vee z_i$, donde $z_i \in L$. La manera en que se construirán C'_j y L'_j depende del valor que tenga i:

Caso 1: $i = 1$

$$L'_j = \{y_j^1, y_j^2\}$$

$$C'_j = (z_1 \vee y_j^1 \vee y_j^2) \wedge (z_1 \vee y_j^1 \vee \overline{y}_j^2) \wedge (z_1 \vee \overline{y}_j^1 \vee y_j^2) \wedge (z_1 \vee \overline{y}_j^1 \vee \overline{y}_j^2)$$

Caso 2: $i = 2$

$$L'_j = \{y_j^1\}$$

$$C'_j = (z_1 \vee z_2 \vee y^1_j) \wedge (z_1 \vee z_2 \vee \overline{y}^1_j)$$

Caso 3: $i = 3$

$$L'_j = \emptyset$$
$$C'_j = C_j$$

Caso 4: $k > 3$

$$L'_j = \{y^k_j \mid 1 \leq kleqi - 3\}$$

$$C'_j = (z_1 \vee z_2 \vee y^1_j) \wedge \{(\overline{y}^k_j \vee z_{k+2} \vee \overline{y}^{k+1}_j) \mid 1 \leq kleqi - 4\} \wedge (\overline{y}^{i-3}_j \vee z_{i-1} \vee z_i)$$

La demostración de que la transformación es válida y de tiempo polinomial se puede ver en [Garey, Jhonson (1979)].

Problema 5 *ACOPLAMIENTOS TRIDIMENSIONALES DE VERTICES* **(atv)**

Sea $M \subseteq W \times X \times Y$, *donde* W, X *y* Y *son conjuntos ajenos tales que* $| W | = | X | = | Y | = q$

Pregunta: *¿*M *contiene un acoplamiento, tal que un subconjunto* $M' \subseteq M$, *donde* $| M' | = q$ *y dos elementos de* M' *no están en una misma coordenada?*

Problema 6 *CUBIERTA DE VERTICES* **(cv)**

Sea $G = (V, A)$ *una gráfica y* $k \leq | V |$ *un entero positivo.*

Pregunta: *¿Existe una cubierta de vertices menor o igual a* k *en* G, *es decir. existe* $V' \subseteq V$ *tal que* $| V' | \leq k$ *y para cada arista* $(u, v) \in A$, *al menos* $u \in A$ *o* $v \in A$*?*

Problema 7 *PARTICION* **(par)**

Sea A *un conjunto, definimos una función*

$$t : A \rightarrow \mathbb{Z}^+$$
$$a \longmapsto t(a)$$

Pregunta: *¿Existe un subconjunto* $A' \subseteq A$ *tal que*

$$\sum_{a \in A} t(a) = \sum_{a \in A - A'} t(a)?$$

Problema 8 *CIRCUITO HAMILTONIANO* **(ch)**

Sea $G = (A, V)$ *una gráfica*

Pregunta: *¿Contiene* G *un* Circuito Hamiltoniano, *es decir, existe un ordenamineto* $<v_1, v_2, v_3, ..., v_n>$ *de vértices de* G, *donde* $| V | = n$, *tal que* $\{v_n, v_1\} \in A$ *y* $\{v_i, v_{i+1}\} \in A$ *para todo* i, $1 \leq i \leq n$?

Problema 9 *BIN PACKING* **(bp)**

Sea $U = \{u_1, u_2, u_3, ..., u_m\}$ *un conjunto de objetos, definimos una función*

$$s : U \to \mathbf{Z}^+$$
$$u \longmapsto s(u)$$

además una constante fija $B \in \mathbf{Z}^+$ *y una variable* $k \in \mathbf{Z}^+$

Pregunta: *¿Existe una partición* $U_1, U_2, U_3, ..., U_k$ *de* U *tal que*

$$\sum_{U_i} \sum_{u \in U_i} s(u) \leq B?$$

Problema 10 *PROBLEMA DE LA MOCHILA* **(pm)**

Sea $U = \{u_1, u_2, u_3, ..., u_m\}$ *un conjunto objetos, definimos una función*

$$s : U \to \mathbf{Z}^+$$
$$u \longmapsto s(u)$$

también definimos una función

$$v : U \to \mathbf{Z}^+$$
$$u \longmapsto v(u)$$

y dos constantes $B, k \in \mathbf{Z}^+$.

Pregunta: *¿Existe un subconjunto* $U' \subseteq U$ *tal que*

$$\sum_{u \in U'} s(u) \leq B \text{ y } \sum_{u \in U'} v(u) \geq k?$$

Problema 11 *PROGRAMACION LINEAL ENTERA* **(ple)**

Sean m, n enteros positivos, $b \in \mathbf{Z}^m$, $c \in \mathbf{Z}^n$, A es una matriz de $m \times n$ con elementos $a_{i,j} \in \mathbf{Z}$. Sea S el conjunto de vectores

$$S = \{x \mid x \in \mathbf{Z}^n, Ax \leq b, x \geq 0\}$$

y una función objetivo definida de la siguiente manera:

$$z : (S, c) \to \mathbf{Z}$$
$$(x, c) \longmapsto z(x, c) = cx$$

también una constante B en $\mathbf{Z}$

Pregunta: *¿Existe $x \in S$ tal que $cx \leq B$?*

Problema 12 *PROBLEMA DE ASIGNACION DE HORARIOS* **(ph)**

Sea $T = \{t_1, t_2, t_3, ..., t_n\}$ un conjunto de tareas, donde cada tarea t_i tiene asignanda una duración $l(t_i)$, un número $m \in \mathbf{Z}^+$ de procesadores, un orden parcial $\preceq$ en T y un plazo $D \in \mathbf{Z}^+$.

Pregunta: *¿Existe una asignación σ de trabajos en los procesadores para T, tal que no exceda el plazo y además no se viole la restricción de precedencia, i.e. si $t \preceq t'$, entonces*

$$\sigma(t') \geq \sigma(t) + l(t) \text{ y } \sigma(t) + l(t) \leq D?$$

4. PROBLEMAS DE BÚSQUEDA

En esta parte mostraremos algunas de las técnicas para el diseño de algoritmos, con el fin de mostrar lo importante que esto puede ser para encontrar exitosamente la solución de un problema, y además, porque en estas técnicas se basan los procedimientos que se mostrarán en el capítulo cinco. Posteriormente introduciremos el concepto de *problemas de búsqueda* **(pb)**, el cual lo tomaremos de *[Garey, Johnson(1979)]*. La propuesta que desarrollaremos es que cualquier problema de decisión se puede plantear como un problema de búsqueda, lo que también implicará que cualquier problema de optimización combinatoria se podrá plantear de esta manera; para terminar se daran las primeras aproximaciones (consideradas de fuerza bruta) de como resolver éste tipo de problemas.

4.1 Algunos Intentos

A través de los años se han generado diferentes técnicas para diseñar algoritmos que sean capaces de resolver una variedad grande de problemas. Algunas de las más importantes son: *divide y venceras* **(dv)**, *algoritmos glotones* **(ag)** y *búsqueda local* **(bl)**. Pero cuando se quiere resolver un problema es importante responder ¿qué tipo de solución nos darán las diferentes técnicas mencionadas?. Además, también es importante saber que tipo de problema es el que estamos tratando de resolver, es decir, si el problema es de la clase **P**, **NP-C** o ninguno de los anteriores.

Sin embargo, hemos visto que para problemas **NP-C** no es posible (todavía) encontrar en tiempo polinomial soluciones para toda instancia del problema, por lo que es conveniente determinar si la instancia del problema tiene características especiales que puedan ser explotadas para poder obtener una solución o, en dado momento, una solución aproximada que pueda ser usada en lugar de la solución óptima.

4.1.1 Divide y venceras

Es una técnica donde se resuelve un problema dividiendolo en problemas pequeños e indepedientes, resolviendo cada subproblema de manera recursiva

para finalmente hacer una combinación de las soluciones y obtener una solución del problema original. La aplicación efectiva de esta técnica depende de la habilidad que se tenga para combinar eficientemente las soluciones de los subproblemas.

En esta técnica se resalta la diferencia entre que el algoritmo sea correcto y eficiente, si el método para combinar la solución no se afecta con la forma en que el problema se dividió en subproblemas, entonces cualquier partición de la instancia original tiene como resultado un algoritmo correcto. Sin embargo, los diferentes métodos para particionar un problema dan como resultado diferentes tiempos de corrida. La eficiencia del paso de combinación está en función del tamaño de la solución de los subproblemas, la cual, típicamente, toma un tiempo lineal.

Pero también existen problemas, tales como el **pav**, para el cual, la combinación de las soluciones de los subproblemas de la instancia original no es tan simple como lo sería tratar de resolver el problema por algún método de fuerza bruta, ver más adelante. En tales casos es coveniente intercambiar optimalidad por eficiencia y tratar de recombinar las soluciones para obtener una solución global aproximada.

Esta forma de diseñar algoritmos nos puede ser de utilidad para construir heurísticas (de momento diremos que una heurística es un algoritmo que no encuentra necesariamente una solución óptima de un problema, pero que es capaz de encontrar una buena solución en tiempo polinomial, para aclarar mejor esto ver capítulo cinco) para los problemas **NP-C**, aunque lo que estaríamos haciendo realmente sería diseñar algoritmos que nos dan una solución aproximada.

Ejemplo 6 *Consideremos el problema del agente viajero euclidiano* **pave**, *en el cual las ciudades son puntos del plano bidimensional y el objetivo es encontrar un recorrido que pase por cada uno de los puntos una sola vez y sea de longitud mínima.*

La figura 4-1 muestra la técnica de divide y venceráz, considerando particiones de cuatro.

En este problema el paso de la combinación de las soluciones locales es fundamental, incluso con una buena división del problema. Los pasos básicos del algoritmo son:

1. Particionar el rectángulo (el cual debe ser el más pequeño que contenga todos los puntos, incluso en su frontera, de la instancia del **pave**, *y que cumpla con que sus aristas son pararelas a los ejes) horizontal y verticalmente en dos subconjuntos tales que:*

i. la dirección de la línea divisora es paralela al lado más corto,

ii. la línea divisora pasa a través de un punto, localizado en dos subproblemas que contienen un número igual de puntos.

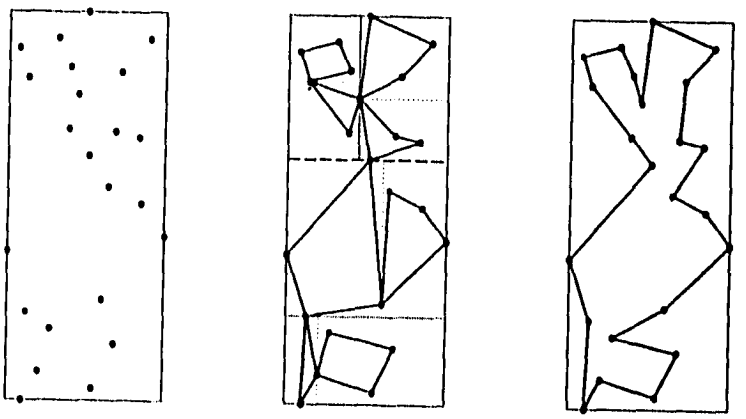


figura 4-1 a) El conjunto de puntos, b) La última partición hecha por el algoritmo, c) El recorrido final

2. Resolver las dos mitades recursivamente,

3. Combinar los dos subpasos, los cuales se deben intersectar en un sólo punto.

Sin embargo, si la unión de dos subpasos se quiere hacer optimamente será necesario que la combinación requiera la substitución de la mayoria de los dos subpasos, como se muestra en la figura 4-2. Pero, para tratar de evitar en lo posible este problema, usaremos una heurística simple para unir a dos subpasos.

Como se muestra en la figura 4.3 para cada uno de los pares de segmentos $(c_1, c_1'), (c_1, c_2'), (c_2, c_1')$ *y* (c_2, c_2') *considérese el efecto de cambiarlos por los segmentos* $\overline{P_1P_1'}$, $\overline{P_1P_2'}$, $\overline{P_2P_1'}$ *o* $\overline{P_2P_2'}$. *La transformación anterior se usó para la construcción de los pasos de la figura 4.1.*

4.1.2 Algoritmos Glotones

Un *algoritmo glotón* trata de obtener una solución óptima de un problema de optimización combinatoria **(POC)**, ver capítulo dos, por medio de una

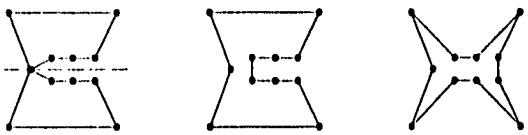


figura 4-2 a) Dos subpaseos vecinos, b) La posible unión de dos subpaseos, c) El paseo óptimo

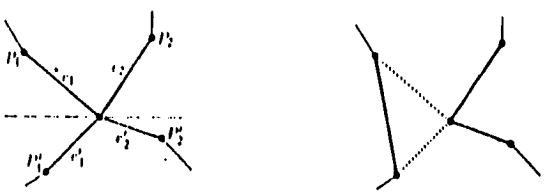


figura 4-3 Una posible combinación

secuencia de elecciones. En cada paso de decisión del algoritmo se hace la mejor elección de ese momento.

Sin embargo, esta estrategia heurística no siempre obtiene la solución óptima de un problema, por lo que no necesariamente se obtiene una solución factible. Pero hay dos características de los problemas que se pueden resolver con esta estrategia; ver *[Cormen, Leiserson, Rivest(1992)]*

Propiedad de Elección Avida: Esta característica se basa en la posibilidad de encontrar una solución óptima a partir de elegir en cada momento la mejor opción que se presente; esta elección depende de las elecciones previas y no depende de las próximas elecciones.

Subestructura óptima: Un problema exhibe esta subestructura si la solución óptima de un problema está formada por la secuencia de las mejores elecciones que se hicieron en el transcurso del algoritmo.

Estas propiedades serán más claras con el siguiente ejemplo, el cual es una variante del **ph**.

Ejemplo 7 *El tipo de* **ph** *que se muestra tiene por objeto realizar la mayor cantidad de tareas, sin violar la restricción de precedencia (de hecho a este problema se le puede llamar de* programación de tareas*). Sea* $T = \{t_1, t_2, t_3, ..., t_n\}$ *un conjunto de n tareas, donde cada tarea* t_i *tiene un tiempo de incicio* s_i *y un tiempo de término* f_i *fijos, donde* $s_i \leq f_i$*. La duración de la i-ésima tarea está definida como el intervalo semiabierto* $[s_i, f_i)$*.*

La precedencia de dos actividades, t_i *y* t_j*, no se viola si sus intervalos de duración no se intersectan, es decir, si* $s_i \geq f_j$ *o* $s_j \geq f_i$*; a las tareas que cumplan con la condición de precedencia se les llamará* compatibles*.*
En estos términos el problema consiste en encontrar el conjunto más grande de tareas compatibles.

El algoritmo glotón que proponen [Cormen et al.] supone que las tareas están ordenadas en forma decreciente con respecto a su tiempo de término, es decir, $f_1 \leq f_2 \leq f_3 \leq ... \leq f_n$ *(si las tareas no estuvieran ordenadas como se mencionó, se pueden ordenar en* $\mathbf{O}(n \log n)$ *con el* quicksort*, [Aho, Hopcroft, Ullman (1974)]; además se rompen los empates arbitrariamente).*

Las estructuras de datos que usa el algoritmo son dos arreglos que contienen los tiempos de inicio y término s y f para cada tarea, el pseudo código del algoritmo se presenta en seguida:

ag-tareas(s, f)

01. $n := length[s]$

02. $A := \{1\}$

03. $j := 1$

04. **for** $i = 2$ **to** n **do**

    05. **if** $s_i \geq f_j$ **then**
    06. $A := A \cup \{j\}$

07. $j := i$

08. **return** A

La operación del algoritmo se muestra en la *figura 4-4*. El conjunto A esta formado por las tareas seleccionadas. La variable j especifica la última tarea agregada al conjunto A, además como las tareas se ordenaron de manera decreciente, con respecto al tiempo de término, entonces f_j siempre será el máximo tiempo de cualquier tarea de A, es decir

$$f_j = \max\{f_k \mid k \in A\}$$

Las líneas 02-03 seleccionan a la tarea 1 (por lo que inicialmente el conjunto A sólo contiene a esta) y se inicializa a j con la tarea 1. Las líneas 04-07 consideran a toda tarea i e incluyen a i en A siempre y cuando ésta última sea compatible con todas las otras tareas que están en A, esto se hace en el paso 05 comparando el tiempo de inicio de la tarea i con el tiempo de término de la última tarea j introducida en A. Si la tarea i es compatible con las demás, los pasos 06-07 introducirán a i en A y actualizaran a j con i.

Este algoritmo es eficiente en el sentido que para n tareas encuentra una solución en tiempo $\mathbf{O}(n + n \log n)$

4.1.3 Búsqueda Local

El algoritmo de búsqueda local **(b)**I necesita del concepto de *vecindad de una solución*, el cual lo podemos definir como:

Definición 23 *Sea* (Γ, k), *una instancia de un* **POC** Π, *donde* Γ *es el conjunto de soluciones de* Π *y* k *es una función que a cada solución* $\gamma \in$ *Gamma le asocia un número real,* una estructura de vecindad *es una función*

$$V : \Gamma \to \wp(\Gamma)$$
$$\gamma \mapsto V(\gamma) \subseteq \Gamma$$

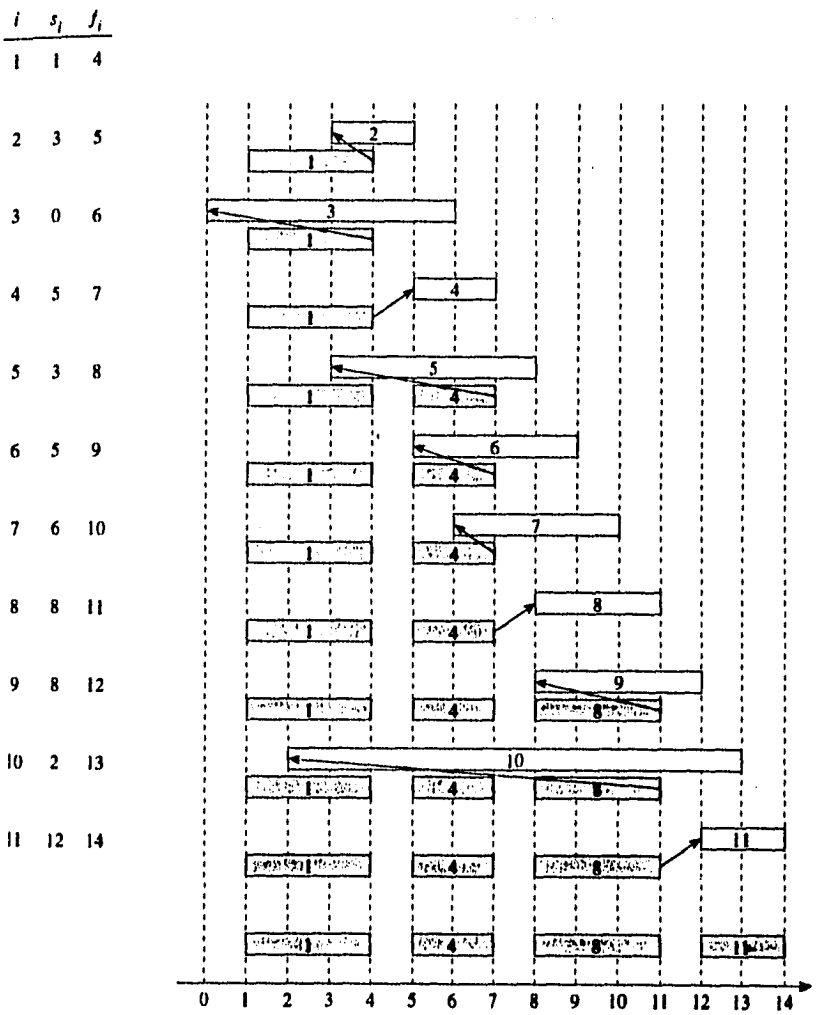


figura 4-4 La ejecución del algoritmo glotón con un conjunto de 11 tareas, mostradas a la izquierda. Cada columna de la figura corresponde a una iteración del for en las líneas 04-07. Las actividades que están en A están sombreadas, mientras que las actividades en blanco están siendo consideradas. Si el tiempo inicial s_i de la tarea i es mayor que el tiempo de término f_j de la tarea seleccionada j (la flecha apunta a la izquierda) la tarea i es rechazada en caso contrario (la flecha apunta a la derecha) la tarea es incluida en A

El sentido que toman los elementos del conjunto $V(\gamma)$ con respecto a la solución γ, es el de que son soluciones *cercanas* con respecto a alguna métrica.

El conjunto $V(\gamma)$ es llamado *vecindad de la solución* γ, y para todo $\gamma' \in V(\gamma)$ se le llama *solución vecina de* γ; obviamente se tiene que cumplir que

$$\gamma' \in V(\gamma) \Leftrightarrow \gamma \in V(\gamma')$$

Bajo este contexto, lo que nos interesará es poder escoger un vecino para $\gamma \in \Gamma$ que sea de cierta manera el adecuado; sin embargo para poder lograrlo tenemos que generar la vecindad de $V(\gamma)$, por lo que nos será útil la siguiente definición:

Definición 24 *Sea* (Γ, f) *una instancia de un* **POC** *y sea* $V(\gamma)$ *una vecindad para la solución* γ, *un* mecanismo de generación *será aquel que selecciona una solución* γ' *de la vecindad* $V(\gamma)$

Se puede argumentar que, dada una instancia de un **POC** y una estructura de vecindad, un algoritmo de búsqueda local itera en un número finito de soluciones, empezando con una solución inicial (que puede ser escogida de acuerdo al problema, o tal vez en forma aleatoria) y, con la aplicación de un mecanismo de generación trata de encontrar una solución mejor (de menor o mayor costo, según sea el caso) buscando en la vecindad de la solución actual. Sise encuentra una solución mejor, la solución actual se reemplaza por esa. En caso contrario, el algoritmo debe continuar con la solución actual. El algoritmo termina si no encuentra mejoras que sean estrictas.

Usando pseudocódigo el algoritmo se puede escribir como:

1. $\gamma := \gamma_0$
2. **repeat**

   este procedimieto genera $\gamma' \in V(\gamma)$

   3. *generar*$[\gamma']$
   4. **if** $f(\gamma') < f(\gamma)$ **then**
   5. $\gamma := \gamma'$

until $f(\gamma') \geq f(\gamma)$ para todo $\gamma' \in V(\Gamma)$

Para aplicar éste método a un problema particular se tienen que considerar varios aspectos. Primero, se debe de decidir cómo obtener una solución inicial factible; en ocasiones es bueno ejecutar el algoritmo **bl** desde varios puntos iniciales y escoger el mejor resultado y en este caso se tiene que decidir cuántos puntos iniciales se pueden utilizar y cómo se deben de elegir, ver *[Papadimitriou, Stiglitz (1982)]*.

Segundo, se tiene que seleccionar una *vecindad* adecuada para el problema y un criterio para elegir un elemento de ésta. La elección usualmente se guía por la intuición y las características del problema, porque no existe teoría que esté disponible para cualquier tipo de problemas de optimización combinatoria.

Consideremos en seguida la ejecución del **bl** para una instancia del **pav** que se muestra en la *figura 4-5*

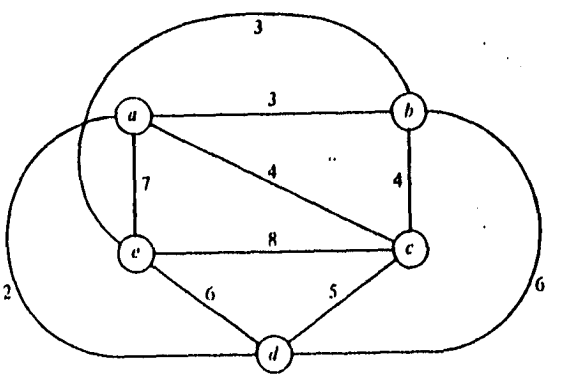


figura 4-5 Una instancia del pav

Ejemplo 8 *Supóngase que se encontró una solución para la instancia del* **pav** *de la figura 4-5 la cual es representada por la figura 4-6(a). Podemos reemplazar las aristas* (a, e) *y* (c, d) *con un costo total de 12 por las aristas* (a, d) *y* (c, e)*, con un costo total de 10, esta operación se muestra en la figura 4-6(b).*

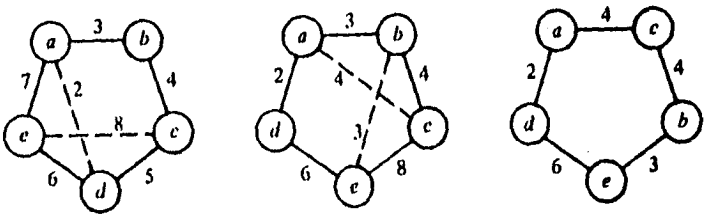


figura 4-6 La optimización de la instancia del pav

Si además reemplazamos las aristas (a, b) *y* (c, e) *por* (a, c) *y* (b, e), *tendremos como resultado la solución óptima que se muestra en la figura 4-6(c). Se puede verificar que ningún par de aristas se pueden quitar de la figura 4-6(c), para mejorar la solución.*.

En la siguiente sección del trabajo se definirá a los problemas de búsqueda, que serán las estructuras con las que se trabajaraá en el resto del mismo, también se mostrarán las primeras aproximaciones que los intentan resolver basándose en los algoritmos de búsqueda en gráficas.

4.2 Problemas de Búsqueda

Para poder empezar a hacer un análisis de los métodos de búsqueda, tenemos que considerar primero que existen diferentes tipos de problemas que abordaremos aquí. Por un lado los problemas que surgen de las aplicaciones de la Teoría de las Gráficas, los cuales tienen en forma implícita un recorrido (o exploración) en su estructura, y los problemas de búsqueda que pueden ser representados por medio de una gráfica.

Del tipo de problemas, que se acaban de mencionar, sólo consideraremos los problemas que sean **NP-C** o **Intratables**. En este tipo de problemas se puede hacer un recorrido para llegar a un estado *conveniente* o final (*i.e.* un estado que represente una solución o que cumpla con las condiciones requeridas), este estado estará representado por un vértice, mientras las relaciones de costo o *dificultad* que se tengan de un estado a otro serán representadas por arcos o aristas.

Uno de los métodos básicos de la representación de problemas es la construccion de un espacio de soluciones en forma de una gráfica, donde los nodos o vértices representan estados y los arcos o aristas representan operadores que transforman cierto estado en otro dentro del mismo espacio de estados.

La idea general es ir buscando una secuencia de estados que bajo alguna *operación* nos lleven de un *estado inicial* a un *estado final.* En este punto, el hecho de que el problema sea **NP-C** o **NP** implica que el número de posibles estados puede ser muy grande.

Si la solución de un problema se basa en encontrar una sucesión de operadores que transformen un estado inicial a un estado final, entonces esto corresponde al problema de gráficas de hallar un camino de un vértice v_0 a otro v.

Como ejemplo de la representación de problemas de búsqueda, consideremos el caso para el problema de programación lineal entera **pple**, definido también por

$$\left.\begin{array}{l} \min z = cx \\ Ax \leq b \\ x \in Z^n \end{array}\right\} \qquad (2)$$

Denotemos el conjunto de soluciones factibles como

$$\Gamma = \{x \mid Ax \leq b, x_i \geq 0, \forall i = 1...n\}$$

Una manera para resolver el problema de manera directa sobre el conjunto Γ, es particionándolo sucesivamente en subconjuntos cada vez más pequeños con la propiedad de que cualquier solución óptima está dada en al menos uno de los subconjuntos.

Esta *división* se puede ilustrar por un árbol numerado como en la *figura 4-7*, donde cada vértice del árbol corresponde a un subproblema de *(2)*, es decir, para el vértice k tenemos:

$$\min z, x \in \Gamma_k \qquad (3)$$

donde

$$\Gamma_k \subset \Gamma$$

Como la numeración de los vértices procede de arriba hacia abajo del árbol, los conjuntos $\Gamma_j, j = 1,...,m$ se hacen más pequeños en forma progresiva hasta que es posible resolver *(3)* o al menos determinar si contiene soluciones potenciales, este método tiene el nombre de *ramificación y acotamiento*, el cual se especificará más adelante.

La numeración procede hasta alcanzar el mínimo de *(2)* pero esto puede tardar demasiado, pués este tipo de problemas son **NP-C**, ver *[Neunhauser(1988)]*.

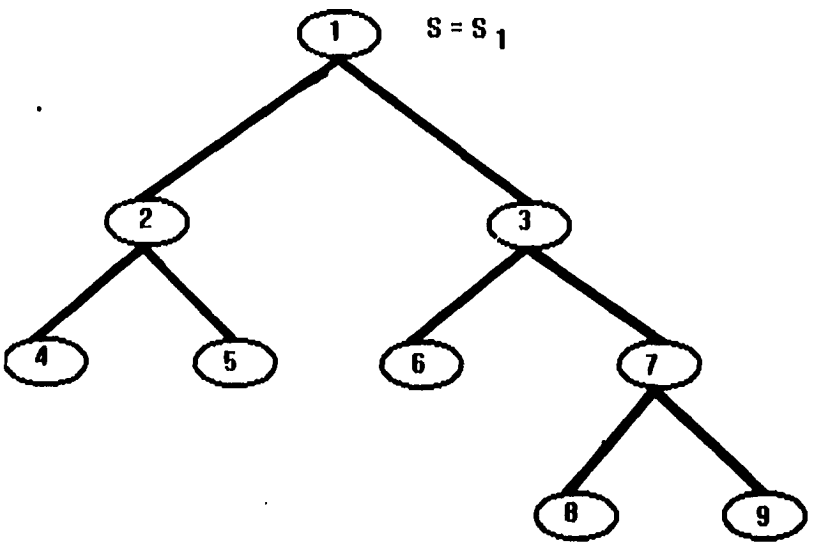


figura 4-7 Representación del árbol generado por la división del conjunto de soluciones factibles

Con todo esto, el objetivo de las estrategias de búsqueda es decidir cual de los operadores (aristas de la gráfica) deben ser incluidos en el camino que da la solución.

La clase de problemas que definiremos incluye las ideas que se han mencionado, y a los elemnetos de esta clase de problemas se denominará de búsqueda **(pb)**.

Un **pb** Π consiste de un conjunto D_Π de instancias y para toda $I \in D_\Pi$ existe un conjunto $\Gamma_\Pi[I]$ de objetos finitos llamados *soluciones* de Π. Diremos que un algoritmo resuelve un **pb** Π si para alguna instancia $I \in D_\Pi$ dada, se obtiene la respuesta *no* siempre y cuando $\Gamma_\Pi[I]$ es vacio, en otro caso, se obtiene como salida una solución γ que pertenece al conjunto $\Gamma_\Pi[I]$.

Por ejemplo, el conjunto de soluciones para una instancia del **pav** consiste de todos los recorridos que tengan el menor costo. Por lo que un algoritmo que pretenda resolver la instancia sólo debe de encontrar una de estas soluciones.

Es importante observar que un problema de decisión Π se puede formular como un problema de búsqueda tomando la siguiente relación:

$$\text{si } I \in Y_\Pi, \text{ entonces } \Gamma_\Pi[I] = \{si\}$$

y

$$\text{si } I \notin Y_\Pi, \text{ entonces } \Gamma_\Pi[I] = \emptyset$$

En el capítulo dos habíamos visto que era posible interpretar a los problemas de optimización combinatoria como problemas de decisión, por lo que se infiere que un problema de optimización combinatoria se puede caracterizar como problema de búsqueda.

Sabemos que para los problemas **NP** no existe un algoritmo determinístico de tiempo polinomial que sea capaz de resolverlo; sin embargo, lo que haremos a lo largo de esta exposición es tratar de aprovechar las características que surgen al representar un **POC** como un problema de búsqueda, para tratar de obtener buenas soluciones en un tiempo razonable. Lo anterior puede ser considerado como nuestra *heurística* para atacar a los problemas **NP**.

Ahora que sabemos que lo que se tiene entre manos son problemas de búsqueda, debemos pensar:

1. ¿cómo hacer una representación del problema?
2. ¿cómo buscar una buena solución γ en esta representación?.

Nuestra propuesta para responder la primer pregunata es la siguiente: si a un **POC** Π se le asocia implícitamente una gráfica conexa, a la que llamaremos *espacio de estados* o *gráfica del problema*, la cual se caracteriza de la siguiente manera:

$$G_\Pi = (E_\Pi, A_\Pi)$$

con un vértice $v_0 \in E_\Pi$ que denominaremos *estado inicial*, donde

$$E_\Pi = \{x \mid x \text{ es un estado de } \Pi\}$$

y las aristas se obtienen usando la estructura de vecindad sobre los posibles estados del problema

$$V : E_\Pi \to \wp(E_\Pi) \text{ tal que}$$
$$x \longmapsto V(x) \in \wp(E_\Pi)$$

de esta manera todas las aristas de G_Π son de la siguiente manera:

$$A_\Pi = \{(x, y) \mid x \in E_\Pi, y \in V(x)\}$$

Además denotaremos a $\gamma \in E_\Pi$ como un estado final, el cual representará una solución del problema.

Si existe un camino de v_0 a γ en la gráfica G_Π, entonces ese camino es una secuencia de pasos (o iteraciones) que nos llevará a una solución de Π.

Sin embargo, de todos los caminos que generan una solución para el problema Π, necesitamos tomar el mínimo (pues necesitamos llegar a una solución en el menor número de iteraciones posibles), para ello necesitamos las siguientes definiciones:

Sea $\Gamma = \{\gamma \mid \gamma$ es una solución de $\Pi\}$ y $C_\Gamma = \{x\gamma\text{-}camino \mid \gamma \in \Gamma, x \in E_\Pi\}$. Definimos la siguiente métrica que nos da la distancia (costo) de un $x\gamma$-*camino*

$$k : C_\Gamma \to \Re^+$$
$$x\gamma\text{-}camino \mapsto k(x\gamma\text{-}camino)$$

Usualmente el cálculo que se hace de $k(xy\text{-}camino)$ se logra por medio de una función de costo c para cada arista de la gráfica, es decir:

$$c : A_\Pi \to \Re^+$$
$$(x, y) \mapsto c(x, y)$$

De esta manera $k(xy\text{-}camino)$ se define como:

$$k(xy\text{-}camino) = \sum_{(u,v)\in xy-camino} c(u, v)$$

necesitamos encontrar

$$\min\{k(x\gamma\text{-}camino) \mid x = v_0\},$$

es decir, queremos encontrar un $v_0\gamma^*$-*camino* que cumpla con,

$$k(v_0\gamma^* - camino) \leq k(v_0\gamma - camino); \forall\gamma \in \Gamma$$

Lo que acabamos de mostrar que un problema de búsqueda se puede reducir a un problema de optimización combinatoria, lo que significa que tenemos una equivalencia entre los tres diferentes tipos de problemas, los de optimización combinatoria, los de decisión y los de búsqueda.

Una vez que se tiene esta estructura (el espacio de estados), podemos pasar a la respuesta de la segunda pregunata, (¿cómo buscar en la representación?). Las estrategias de búsqueda difieren en la cantidad de conocimiento que usan. Se examinarán estrategias las cuales no usan información (primero en amplitud y primero en profundidad), y algunas consideradas heurísticas.

Supongamos el siguiente ejemplo en el cual queremos encontrar el camino más corto en un espacio de estados del estado s al estado g, ver *figura 4-8.*

Entonces podemos representar su espacio de búsqueda como lo muestra la *figura 4-9.*

En este espacio de búsqueda de todos los caminos de s a g sólo nos interesará el camino cuya distancia sea la mínima.

4.3 Estrategias de Fuerza Bruta

En la formulación de los espacios de estado de los problemas, una solución es obtenida por la aplicación de operadores a los estados hasta que una expresión describa algún estado terminal. En general las estrategias (métodos)

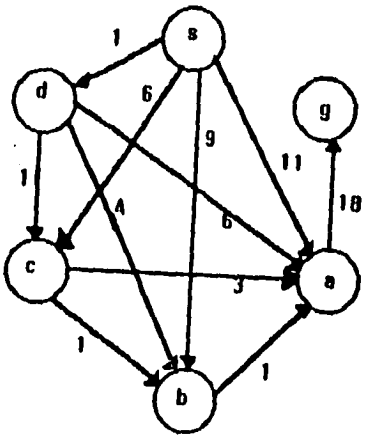


figura 4-8 Un espacio de estados

de búsqueda para los espacios de estados deben de poder considerar los siguientes aspectos.

1. Un vértice inicial asociado con una descripción del estado inicial.
2. Los sucesores de un estado v son calculados usando el operador que es aplicable para la descripción del estado asociado con v. Sea V el operador especial que calcula todos los sucesores de v (los estados que esten en la vecindad de v). Al proceso de aplicar V se le llamará *expandir el estado.*
3. Un conjunto de apuntadores de cada sucesor hacia su predecesor. Estos apuntadores indican el camino de regreso hacia el estado inicial, cuando el estado meta se ha encontrado.
4. Los estados sucesores son examinados para ver si son estados terminales. Y esto se hace hasta encontrar uno. Cuando un estado terminal es encontrado los apuntadores son trazados de regreso hasta el estado inicial para producir el camino que da la solución.

Las características anteriores sólo describen la mayoría de los elementos de las estrategias de búsqueda. Una descripción completa de una estrategia de

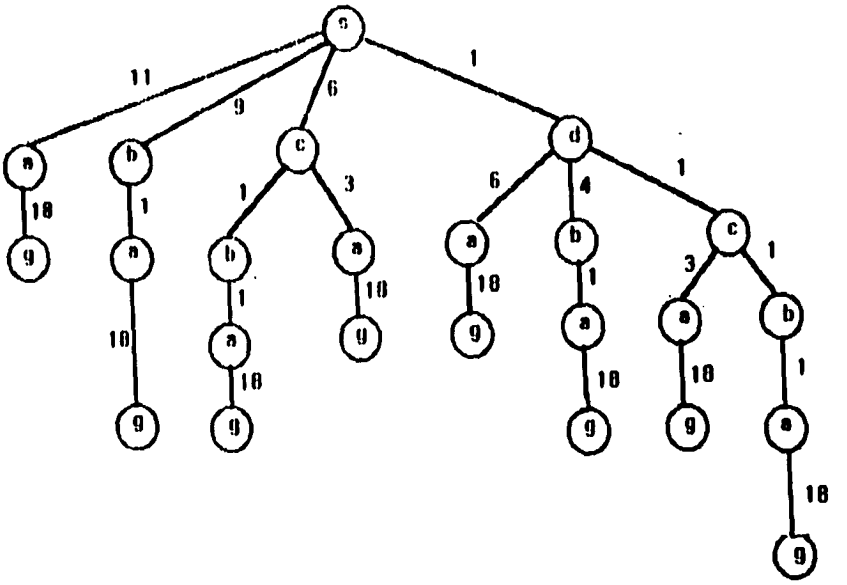


figura 4-9 Representación de el espacio de búsqueda

búsqueda debe describir el orden en el cual los estados deben ser expandidos. La manera inmediata es considerar la idea de los algoritmos de **BP** y **BA** (ver Anexo 1) para la exploración de gráficas.

Consideraremos en nuestras estrategias de búsqueda, la caracterización de los estados en dos clases, que es similar a la que tienen los algoritmos de búsqueda en gráficas, ya que a los vértices que llamaremos *abiertos* y *cerrados* son semejantes a los vértices pintados de gris y negro de los algortimos de búsqueda.

- *Abiertos:* Son los estados que ya han sido generados (y para algunos *algoritmos heurísticos* ya les fué aplicada la función heurística), pero que todavía no han sido considerados (examinados). Esta lista en realidad es una cola de prioridades (en esta cola los elementos con mayor prioridad son aquellos que tienen un valor más promisorio obtenido por la función heurística).

- *Cerrados:* Los estados que ya han sido examinados.

4.3.1 Estrategia de Búsqueda Primero en Amplitud.

Esta estrategia de búsqueda expande los estados en el orden en el cual ellos son generados. La estrategia consiste de los siguientes pasos.

1. Colocar el estado inicial en *abiertos*
2. Si *abiertos* está vacía, salir con una falla
3. Remover el primer estado de *abiertos* y ponerlo en *cerrados*, llamar a este estado v
4. Expandir v, generando todos sus sucesores; si no tiene sucesores, ir a (2). Sino, poner los sucesores al final de *abiertos* y colocar los apuntadores de estos sucesores hacia v,
5. Si alguno de sus sucesores es un estado terminal (meta) salir con la solución, obteniéndola a través de los apuntadores, en caso contrario ir a (2).

Este algoritmo supone que el estado inicial no es terminal (meta). Se puede garantizar que esta estrategia encuentra un camino solución, si es que existe, de v_0 a un estado terminal, porque en el peor de los casos explorará todo el espacio de estados.

4.3.2 Estrategia de Búsqueda Primero en Profundidad.

En este tipo de estrategia se expande el primero de los estados generados recientemente. Además como en el **BP** se define la *profundidad* de los estados como sigue:

La profundidad del estado inicial es cero. La profundidad de cualquier otro estado diferente al estado inicial es uno más la profundidad de su predecesor (padre).

Para seleccionar un estado del espacio se tomará en cuenta el estado que tenga mayor profundidad. Realizándose siempre y cuando se esté en un camino que resulte fructífero, de lo contrario, es necesario regresar al predecesor de este estado y continuar con la búsqueda.

También puede considerarse el criterio de que si un estado sobrepasa alguna cota de profundidad y no se ha encontrado hasta ese momento una solución se continuará la búsqueda con otro estado que no la sobre pase, pero en caso

contrario la búsqueda se detiene. Este criterio resta el sentido exhaustivo que tiene este método, e incluso se puede ver como un método heurístico. A continuación se define la estrategia:

1. Colocar el estado inicial en *abiertos*
2. Si *abiertos* esta vacía, salir con una falla
3. Remover el primer estado de *abiertos* y ponerlo en *cerrados*, llamar a esta estado v
4. Si la profundidad de v es igual a la cota de profundidad ir a (2)
5. Expandir v, generando todos sus sucesores; poner los sucesores al inicio de *abiertos* y colocar los apuntadores de estos sucesores hacia v.
6. Si alguno de sus sucesores es un estado terminal (meta) salir con la solución obteniéndola a través de los apuntadores; en caso contrario ir (2).

Notemos que el algoritmo numera los estados en el orden en que los expandió. También la estrategia búsca en un camino hasta encontrar una solución o llegar a la cota de profundidad; en tal caso empieza a considerar caminos alternativos de la misma o menor profundidad con respecto al paso anterior, si tampoco aquí encuentra una solución, entonces busca en el predecesor de su predecesor y así sucesivamente.

Los métodos expuestos son especializaciones de los algoritmos **BA** y **BP** los cuales exploraban cualquier tipo de gráfica. De tal manera el comportamiento de las estrategias específicas es semejante.

Pero a pesar de que las estrategias de búsqueda son eficientes porque encuentran una solución, si es que existe, de un problema II, estas pueden ser muy costosas si la representación de II necesita de una cantidad suficientemente grande de estados; o también si se tiene que buscar en todo un espacio de soluciones. Un problema *ad hoc* que muestra lo anterior, es el problema de encontrar una estrategia óptima para el juego de ajedrez, en el cual se ha visto, que el número promedio de posibles movimientos por tirada para cada jugador es de 35 y el número promedio de posibles tiradas en una partida completa es de 50 para cada jugador, ver *[Winston(1977)]*, por lo que si quisieramos construir el espacio de estado para este problema nos encontrariamos con un árbol *35-regular* de 100 niveles (50 para cada jugador), y haciendo cuentas tenemos que el espacio de busqueda constaría de 35^{100} estados, lo que se considera como un problema intrtatable.

4.3.3 Estrategia de Búsqueda Optima

Cuando se ve la verdadera magnitud de las cantidades anteriores, se debe de pensar en métodos que sean capaces de dar una buena solución sin que tengan que explorar todo ese espacio, por lo que se requiere de un método capaz de buscar en las opciones que garanticen una solución del espacio de búsqueda.

El primer método de este tipo es el conocido con el nombre de *Estrategia de Búsquda Optima* **(BO)**, el cual sigue la estrategia de los algoritmos de búsqueda local. Este método a pesar de ser considerado en la literatura clásica de Inteliegencia Artificial como de fuerza bruta no lo es estrictamente, porque esta estrategia no tiene que realizar una búsqueda en todo el espacio, ya que usa una función de evaluación que genera un costo por obtener a cada sucesor de todos los estados del espacio; avanzará en la búsqueda tomando el estado sucesor que tenga menor costo. De esta manera en el camino que se va construyendo, para cada estado v que este en él, se podrá garantizar que el costo de v al estado inicial v_0 es el menor.

Considérese el problema, Π_1, de encontrar el camino de costo mínimo del estado inicial v_0 a un estado final γ para el caso en el que una gráfica representa un espacio de estados con los costos de los operadores determinados para todas las aristas por una función definida como:

$$\begin{aligned} c &: A_{\Pi_1} \rightarrow \Re^+ \\ (u,v) &\mapsto c(u,v) \end{aligned}$$

donde

$$A_{\Pi_1} \subseteq \{(u,v) \mid u \in E_{\Pi_1}, v \in V(u)\}$$

El costo del camino del estado v_0 a un estado v se calculará con una función recursiva que se define como:

$$\begin{aligned} g(v_0) &= 0 \\ g(v) &= g(u) + c(u,v) \end{aligned}$$

El algoritmo usa también las lista de *abiertos* y la lista de *cerrados* y se define a continuación:

1. Meter a v_0 en *abiertos* y $g(v_0) = 0$
2. Si *abiertos* $= \emptyset$, salir con falla
3. encuentrar v tal que $g(v) = \min\{g(u) \mid u \in \textit{abiertos}\}$
4. Si $v \in \Gamma$ terminar con éxito

5. Sacar a v de *abiertos* y meterlo a *cerrados*

6. Obtener $V(v)$

7. Para todo $w \in V(v)$ calcular $g(w)$

   (a) Si $w \notin$ *abiertos* y $w \notin$ *cerrados*, entonces v meterlo a *abiertos*. Asignarle $g(w)$ a el estado w,

   (b) Si $w \in$ *abiertos* o $w \in$ *cerrados*, comparar el nuevo valor $g(w)$ asignado a w con el valor previamente asignado, que llamamos $g'(w)$

      i. Si $g'(w) \leq g(w)$, entonces no cambiar nada,

      ii. Si $g'(w) > g(w)$, entonces, substituir $g(w)$ por $g'(w)$ y actualizar el apuntador hacia v,

8. ir a 3.

Se puede onbservar que si el costo de todas las aristas es constante este método es equivalente a la estrategia de Búsqueda en Amplitud.

Ejemplo 9 *Supongamos el siguiente espacio de estados:*

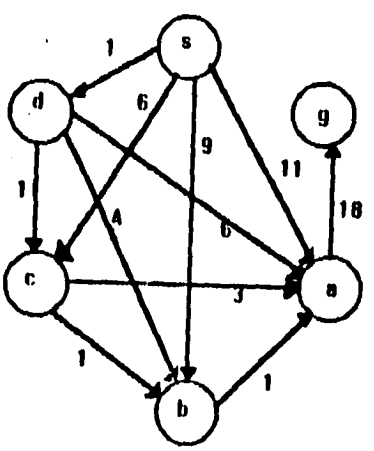


figura 4-10 El espacio de estados

En la tabla se muestran las operaciones del **BO** *sobre la instancia de la figura 9-10*

vértice	*adyacencia*
s	$\bullet \longmapsto a\bullet \longmapsto b\bullet \longmapsto c\bullet \longmapsto d\bullet \longmapsto nil$
a	$\bullet \longmapsto \mathbf{g}\bullet \longmapsto nil$
b	$\bullet \longmapsto a\bullet \longmapsto nil$
c	$\bullet \longmapsto a\bullet \longmapsto b\bullet \longmapsto nil$
d	$\bullet \longmapsto a\bullet \longmapsto b\bullet \longmapsto c\bullet \longmapsto nil$
$\mathbf{g}$	$\bullet \longmapsto nil$

1: $s = v$, $g(s) = 0$, *abiertos* $= \{s\}$, *cerrados* $= \emptyset$

—	v	$v \neq \mathbf{g}$	*abiertos*	*cerrados*	$V(v)$	$g(V(v))$
2:	s	—	$\emptyset$	$\{s\}$	$\{a,b,c,d\}$	$g_s(a) = 11, g_s(b) = 9,$ $g_s(c) = 6, g_s(d) = 1$
3:	d	$d \neq \mathbf{g}$	$\{a,b,c\}$	$\{s,d\}$	$\{a,b,c\}$	$g_d(a) = 7, g_d(b) = 5,$ $g_d(c) = 2$
4:	c	$c \neq \mathbf{g}$	$\{a,b\}$	$\{s,d,c\}$	$\{a,b\}$	$g_c(a) = 5, g_c(b) = 3$
5:	b	$b \neq \mathbf{g}$	$\{a\}$	$\{s,d,b,c\}$	$\{a\}$	$g_b(a) = 4$
6:	a	$a \neq \mathbf{g}$	$\emptyset$	$\{s.d.c.b.a\}$	$\{\mathbf{g}\}$	$g_a(\mathbf{g}) = 22$
7:	$\mathbf{g}$	$\mathbf{g} = \mathbf{g}$	—	—	—	—

Notemos que el camino mínimo está dado por $s - d - c - b - a - \mathbf{g}$ con un peso total de 22.

Sólo falta demostrar que el algoritmo encuentra la solución óptima, y para poderlo hacer consideremos el siguiente teorema.

Teorema 6 *Sea $G_\Pi = (E_\Pi, A_\Pi)$ un espacio de estados de un problema de búsqueda Π, supongamos que existe un $v_0\gamma$-camino en G_Π, entonces el método de búsqueda óptima terminará con exito.*

Demostración. *Supongáse que el procedimiento falla, entonces por el paso (2) la lista de abiertos está vacía, mientras la lista de cerrados contiene a todos los vértices explorados hasta ese momento. Pero como existe $v_0\gamma$-camino, el cual puede ser representado lexicográficamente como $v_0, v_1, v_2, ..., v_n$, entonces debe de haber un $v_i \in v_0\gamma$-camino, además también $v_i \in$ cerrados y fue el último vértice explorado por el algoritmo antes de fallar, entonces $v_{i+1} \notin$ cerrados y $v_{i+1} \notin$ abiertos. Como v_i está en cerrados por el paso (06) tuvo que haber sido expandido, entonces por el paso (07) v_{i+1} tiene que estar en la lista de abiertos al menos una vez y tuvo que ser examinado antes de que la lista de abiertos se vaciara, por lo que se tienen dos casos:*

(a) Si $v_{i+1} \in \Gamma$ tuvo que haber terminado con exito, lo cual es una contradicción.

(b) Si $v_{i+1} \notin \Gamma$ debío de meterse a la lista de cerrados y continuar y como G_Π es finita tendría que terminar hasta que se reduciria al caso anterior, lo cual es una contradicción

La contradicción viene de suponer que el procedimiento falla. Por lo tanto el procedimiento si termina con la solución □

Sin embargo, en general no se puede tener la especificación de los costos en un problema, y para estos casos la estrategia **BO** no es efectiva. Con este antecedente en el próximo capítulo se estudiarán procedimientos de búsqueda heurísticos que lográn en la mayoría de los casos mejorara la eficiencia en la exploración de un espacio de estados.

5. MÉTODOS DE BÚSQUEDA HEURíSTICOS

Inicialmente en este capítulo se dará un ejemplo que muestre una representación del **pav** como problema de búsqueda, resuelto por el método de ramificación y acotamiento. Posteriormente entraremos más a detalle en los métodos de búsqueda heurísticos, empezando con el *Primero Mejor*, y posteriormente con el A^*, el cual es una especialización del anterior.

5.1 Heurística

La noción de la búsqueda heurística apareció en los años 60's. Inicialmente se trataba de aumentar la efectividad de los procesos solucionadores de problemas. Esta herramienta permitía eliminar soluciones del conjunto de todas las posibles, usando el conocimiento disponible acerca del problema para dirigir la búsqueda. Posteriormente las reglas de la búsqueda heurística fueron formalizadas por *[Newell(1969)]* y *[Pearl(1983)]*. Lo que se hará en este capítulo es estudiar estos procesos y tratar de enfocarlos para poder aproximar soluciones de problemas **NP**.

Cuando un problema tiene una representación por un espacio de estados, la búsqueda en el espacio de estados puede hacerse más eficiente si se usa información (o conocimiento) referente al problema.

Diferentes autores definen la búsqueda heurística de varias formas, algunas de ellas son:

- Es una estrategia práctica que incrementa la eficiencia para resolver un problema complejo. *[Feigenbaum, Feldman (1963)]*.
- Llega a una solución a lo largo del camino más probable, omitiendo el menos promisorio. *[Aleksander, Ferreny, Ghallab (1986)]*, *[Amarel(1968)]*.
- Da un simple criterio para seleccionar la dirección de un curso de acción sin determinar de manera definitiva los estados convenientes o inconvenientes *[Guida, Somalvico (1979)]*.
- Es una técnica que mejora la eficiencia de un proceso de búsqueda posiblemente sacrificando la completez. *[Rich(1983)]*.
- Son algoritmos que no tratan de encontrar una solución óptima sino una *buena solución* en un tiempo aceptable. *[Garey, Johnson(1979)]*.

- Al conocimiento que no garantiza estar completamente correcto, pero que es útil para resolver un problema en la mayoría de los casos, se le llama conocimiento heurístico. Un método de búsqueda que usa conocimiento heurístico se llama método de búsqueda heurística. *[Shirai, Tsujii (1982)]*

El uso de heurísticas para incrementar la eficiencia en la búsqueda ha sido estudiado en Inteligencia Artificial (**IA**) y en Investigación de Operaciones (**IO**). En **IO** la representación más común para esta información han sido las funciones de acotamiento que intervienen en la búsqueda con *ramificación y acotamiento*, principalmente para los problemas de programación lineal entera. En **IA** los métodos de búsqueda heurísticos son aplicados a una variedad diversa de problemas, sobre todo en problemas de juegos (o puzzles), planeación de caminos en robótica, o en la obtención de soluciones dadas por sistemas expertos (en campos como la medicina, la biología, la arqueología, entre otros) cuando el espacio de busqueda es muy amplio, *[Jackson(199)]*, *[Pearl(1984)]*, *[Latombe()1991]*. *[Lin (1965)]* propuso el uso de una función de evaluación para dirigir la búsqueda en gráficas y la aplicó para el problema del agente viajero.

Para hacer más concretas estas ideas consideremos el siguiente ejemplo, con el viejo problema del agente viajero:

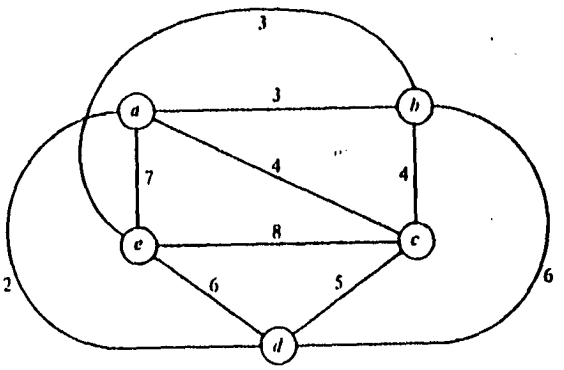


figura 5-1 Una instancia del pav

Ejemplo 10 *Supogamos que tenemos la instancia representada por la gráfica* $G = (V_G, A_G)$ *mostrada en la figura 5.1; la manera en que trataremos de*

determinar una solución aproximada para el **pav**, *es planteándolo como un problema de búsqueda.*

El espacio de búsqueda para el problema estará representado en un árbol cuyo estado inicial representará todos los paseos (recorridos) hamiltonianos. Cada estado tendrá dos hijos, uno de los cuales representará un paseo que contenga una arista determinada, mientras que el paseo del otro estado no la contiene.

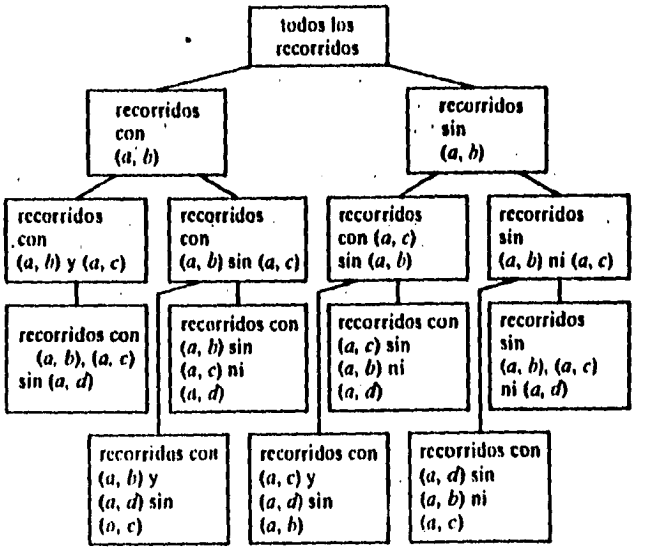


figura 5-2 Principio de un árbol de solución para la instancia del pav

Por las características particulares del **pav** *podremos eliminar a los hijos de algún estado que no represente paseos hamiltonianos; por ejemplo no deben existir estados que representen paseos que contengan a las aristas* $(a,b),(a,c)$ *y* (a,d) *porque a tendría grado 3 y una de las propiedades de los paseos hamiltonianos es que todos sus vértices (o ciudades) sean de grado dos; similarmente, podremos también eliminar estados en los que alguna ciudad tenga grado menor a dos, por ejemplo no se debe encontrar un estado que represente un recorrido sin las siguientes aristas:* $(a,b),(a,c)$ *o* (a,d) *pues sólo se podría llegar a a por* (a,e), *lo que indicaría que a tiene grado uno. (ver figura 5.2)*

La información que necesitamos para poder usar la técnica de ramificación y acotamiento es una cota inferior del conjunto de soluciones del **pav**.

Consideremos a Γ *como el conjunto de soluciones factibles del* **pav**, *obsérvemos que* $\gamma \in \Gamma$ *es un paseo hamiltoniano que no necesariamente es el de longitud*

(costo) mínima; definimos

$$z : \Gamma \to \Re^+ \cup \{0\}$$
$$\gamma \longmapsto z(\gamma)$$

de la siguiente manera

$$z(\gamma) = \sum_{(u,v)\in\gamma} d(u,v) = \frac{\sum_{u\in\gamma, v,w\in ady[u]} (d(u,v) + d(u,w))}{2}$$

Esta función es el costo del ciclo hamiltoniano, donde $d(u,v)$ es la función que determina la distancia entre los vértices u y v y los vecinos de u están representados por:

$$ady[u] = \{x_u \in V_G \mid (u, x_{u,i}) \in A_G\}$$

Observemos que en la última relación los costos de las aristas se cuentan dos veces, por eso dividimos entre dos. Esta igualdad nos será útil para encontrar una cota inferior para el **pav**.

Esto es, sea $u \in \gamma$ un vértice de G, agruparemos a los vecinos de u en orden ascendente con respecto a la distancia con u en la siguiente manera

$$d(u, x_{u,i}) \leq d(u, x_{u,i+1}), i = 1, 2, ...$$

para

$$x_{u,i}, x_{u,i+1} \in ady[u]$$

esto quiere decir que $ady[u]$ es un conjunto ordenado. Entonces

$$d(u, x_{u,1}) + d(u, x_{u,2}) \leq d(u, x_{u,i}) + d(u, x_{u,j})$$

donde $d(u, x_{u,i})$ y $d(u, x_{u,j})$ están en el paseo γ y $d(u, x_{u,1}), d(u, x_{u,2})$ son las dos aristas incidentes a u de menor costo.

Como lo anterior se cumple para todos los vértices del paseo tenemos que

$$\frac{\sum_{u\in\gamma} (d(u, x_{u,1}) + d(u, x_{u,2}))}{2} \leq \frac{\sum_{u\in\gamma} (d(u, x_{u,i}) + d(u, x_{u,j}))}{2}$$

De esta manera la cota inferior para el costo de los paseos del **pav** *está dada por*

$$\frac{\sum_{u \in \gamma} (d(u, x_{u,1}) + d(u, x_{u,2}))}{2}$$

En concreto para la instancia del **pav** *que estamos considerando tenemos que:*

vértice	*aristas*	costo	*total*
a	$(a,b),(a,d)$	$3+2$	5
b	$(b,a),(b,e)$	$3+3$	6
c	$(c,a),(c,b)$	$4+4$	8
d	$(d,a),(d,c)$	$2+5$	7
e	$(e,b),(e,d)$	$3+6$	9
suma *total*			$\frac{35}{2} = 17.5$

Por lo que ningún paseo será menor a 17.5. Es necesario observar que la cota se construyó sin considerar las restricciones del **pav**; *es decir, la suma total (17.5) puede no ser el costo de ningún paseo.*

Ahora, para calcular una cota inferior del costo de los recorridos definidos para algún estado del espacio es necesario considerar que la selección de las dos aristas adyacentes a un vérice u de G se hará de la siguiente manera: como construimos el árbol de tal manera que cada uno de los estados representa recorridos definidos por un conjunto de aristas que deben estar en el paseo y otro conjunto de aristas que no estén, es claro que una arista obligada a permanecer en cualquier paseo debe incluirse entre las dos aristas seleccionadas, sin importar si no es una de las dos aristas de menor costo incidentes a u. Así mismo, una arista restringida a estar fuera de un paseo no puede seleccionarse, aunque su costo sea menor.

Por ejemplo, si existe la restricción de incluir la arista (a,b) y excluir (a,d) y (a,e), las dos aristas para el vértice a son (a,b) y (a,c), con un costo total de 7. Para b se seleccionan (a,b) y (b,e), con costo de 6. Para c se eligen (a,c) y (c,b), con costo de 8. Para d se seleccionan (d,c) y (d,e), con un costo de 11, mientras que para e se deben escoger (e,b) y (e,d) con un costo de 9. La cota inferior para los costos de los paseos que cumplen estas restricciones es $\frac{7+6+8+11+9}{2} = 20.5$.

Para la construcción del árbol de búsqueda a partir de lo que hemos venido mencionando, tenemos que cada vez que se ramifica al considerar los dos hijos de un estado se intenta inferir decisiones adicionales sobre las aristas que deben incluirse o excluirse de los paseos . Las reglas para estas inferencias son:

1. Si la exclusión de una arista (u,v) hiciera imposible para u o v tener dos aristas adyacentes en el recorrido, entonces debe incluirse (u,v).

2. Si incluir (u, v) hace que u o v tengan más de dos aristas adyacentes en el recorrido, o que se complete un ciclo que no sea hamiltoniano con las aristas ya incluidas, entonces se debe excluir a (u, v).

Al ramificar, después de hacer todas las inferencias posibles, se calculan las cotas inferiores de ambos hijos. Si la cota inferior de un hijo es mayor o igual que el paseo de menor costo encontrado hasta el momento, entonces ya no se explora ese hijo y no se construyen sus descendientes (usualmente a esto se le conoce como podar).

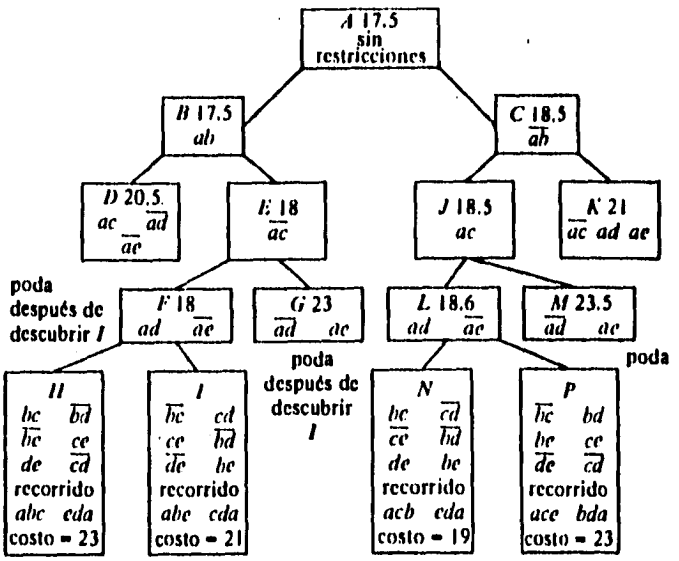


figura 5-3 Arbol de solución para la instancia del pav

Si no es posible podar ningún hijo, como regla heurística se considerará primero al hijo (estado) que tenga la menor cota inferior, con la suerte de alcanzar más rápido una solución de menor costo que la encontrada hasta ese momento. Después de considerar un hijo, se debe de evaluar si puede podarse su hermano, ya que pudo haberse encontrado una solución de menor costo.

Para el caso de nuestra instancia del **pav** *el árbol de búsqueda se muestra en la figura 5-3. Para interpretar los estados del árbol, se debe aclarar que las letras mayúsculas son los nombres de los estados, los números son las cotas inferiores y se listan las restricciones aplicadas al estado, escribiendo uv si la arista (u, v) debe incluirse y $\overline{uv}$ si la arista (u, v) no debe incluirse. Observemos también que las restricciones introducidas en un estado se aplican a todos sus*

descendientes. Así, para obtener todas las restricciones aplicables en un estado debe seguirse el camino de ese estado al estado inicial.

El ejemplo anterior nos muestra dos cosas importantes: una representación del **pav** como problema de búsqueda, usando la filosofía de ramificación y acotamiento, y la necesidad de tener buenos métodos de búsqueda que consideren esta filosofía. Es importante notar que la búsqueda que se hizo en el espacio de estados de la *figura 5-3* sólo se expandieron los estados más prometedores con respecto al valor de sus cotas; este método es el primero del que se hablará en la siguiente sección.

Considerando la definición que da *[Shirai, Tsujii (1982)]* para encontrar la solución del problema en cuestión, necesitamos una función que para cada estado nos diga, a partir del conocimiento considerado, qué tan fructífera sería la búsqueda si lo consideramos. En este punto es importante mencionar que existen problemas de búsqueda en los cuales una solución se va construyendo en una secuencia de estados, como veremos más adelante, por lo que la construcción de la función, a la cual llamaremos *función heurística*, dependerá de cómo es el tipo de espacio de búsqueda. Por ejemplo, para el espacio de estados del **pav** que acabamos de ver, lo que vendría siendo la función heurística es el valor que tenemos de las cotas inferiores en los estados.

Para el caso en el que la solución se va construyendo a través de una secuencia de estados, la función heurística debe generar una aproximación del costo del $x\gamma^*$-*camino*, para todo $x \in E_\Pi$. Dicha función se definirá de la siguiente manera:

$$h : E_\Pi \to \Re^+ \cup \{0\}$$
$$v \longmapsto h(v)$$

Esta función será usada por todos los métodos heurísticos que se verán. Además notemos que para los estados γ, que representen una solución factible de un problema Π, $h(\gamma) = 0$. Esta igualdad es clara por la definición de h.

Los métodos de búsqueda que se analizarán en seguida son métodos de ramificación y acotamiento con una estimación proporcionada por la función heurística en cada estado, siguiendo una modificación del algoritmo de búsqueda local.

5.2 Búsqueda Primero Mejor (PM)

Esta estrategia se usa con la filosofía de *ramificación y acotamiento* y usan las listas de *abiertos* y *cerrados* definidas en el capítulo anterior. El método es muy natural para los problemas de búsqueda, en el sentido de que para cualquier estado podemos obtener a sus sucesores (hijos), y de los hijos de todos los estados considerar sólo el más conveniente con respecto al valor de la función heurística h; es decir, aquél que cumpla:

$$h(y) = \min\{h(x) \mid x \in \textit{abiertos}\}.$$

De esta manera, sólo el estado más prometedor se ramifica de tal manera que la solución se encontrará, si es que existe.

La estrategia **PM** se describe a continuación.

Sea v_0 el estado inicial del problema Π planteado como un problema de búsqueda sobre un espacio de estados $G_=(E_\Pi, A_\Pi)$

1. poner v_0 en la lista *abiertos*, y hacer $h(v_0) = 0$;
2. si la lista de *abiertos* es vacía terminar con falla;
3. sacar de *abiertos* al estado v que cumpla con

   $$h(v) = \min\{h(u) \mid u \in \textit{abiertos}\}$$

   y meter v a *cerrados*;
4. si $v \in \Gamma$ terminar con éxito;
5. determinar $V(v)$
6. para todo $w \in V(v)$ calcular $h(w)$
   (a) si $w \notin$ *abiertos* y $w \notin$ *cerrados*, entonces meterlo a *abiertos*,
   (b) si $w \in$ *abiertos* o $w \in$ *cerrados*, comparar el nuevo valor $h(w)$ asignado a w con el valor previamente asigando, que llamamos $h'(w)$
      i. Si $h'(w) \leq h(w)$, entonces no cambiar nada,
      ii. Si $h'(w) > h(w)$, entonce substituir $h(w)$ por $h'(w)$ y actualizar el apuntador hacia v,
7. ir a 2.

Este algoritmo garantiza en el paso (4) que terminará con éxito siempre y cuando algún estado cumpla con ser el mínimo de todos los estados que están en *abiertos*. Pero este estado puede que no esté necesariamente cerca del óptimo, lo que implica que de esta manera no se puede garantizar la optimalidad de la solución. Sin embargo, tenemos que garantizar primero que el algoritmo sea completo, es decir, termina con una solución, si es que ésta existe.

La razón por la que el algoritmo **PM** no termine con una solución es que la lista de *abiertos* se vacíe en un paso intermedio, lo que significa que no existe un camino de v_0 a algún $\gamma \in \Gamma$.

Para que la lista de *abiertos* se vacíe tiene que pasar que a partir de cierto momento no se metan nuevos estados a la lista, de este modo de todos los estados que estén en la lista de *abiertos* en el paso (3) el algoritmo escogerá al que tenga una mejor estimación y luego en el paso (4) se tendrá que verificar si es un vértice meta.

Además para que no se incremente la lista de *abiertos* es necesario que el conjunto de sucesores sea vacío, así, cada vez que se saque un estado de *abiertos* en el paso (3) se meterá una sóla vez a la lista de *cerrados*, y como las listas son finitas, entonces el algoritmo encontrará que no existe solución. Sin embargo, si existe un $v_0\gamma$-*camino* el algoritmo será capaz de encontrarlo en tiempo finito, por lo que se puede enunciar el siguiente teorema:

Teorema 7 *Sea* $G_\Pi = (E_\Pi, A_\Pi)$ *un espacio de estados para el problema ; si se corre el algoritmo* **PM** *sobre* G_Π, *entonces* **PM** *encontrará la solución.*

Demostración: *La demostración de este teorema es semejante a la demostración del teorema que garantiza que el algoritmo de* **BO** *encuentra la solución, y con lo que se argumentó anteriormente sobre las operaciones de las listas de abiertos y cerrados.* □

Para darnos una idea de la manera en que funciona este método, consideremos a la función heurística como una función obejetivo, definada por el *criterio de Metropolis*, propuesto en *[Metropolis, et al. (1953)]*

$$exp\frac{\Delta E}{k_B T}$$

donde ΔE es el incremento positivo del nivel de energía del estado E_i al estado E_j, T es la temperatura, y k_B es la constante de Boltzmann, en un *sistema físico de multiparticulas.*

Supondremos que podemos usar el algoritmo de *recocido simulado* (**rs**) (*simulated annealing*), propuesto por *[Kirkpatrick, Gellatt, Vecchi (1983)]*, para generar una solución de un problema de optimización combinatoria (**POC**) Π por la analogía de un sistema físico de multiparticulas con un (**POC**), basado en las siguientes relaciones:

1. La soluciones del **POC** son equivalentes a estados del sistema físico

2. El costo de la solución es equivalente a la energía del estado

Se introducirá un parámetro el cual tendrá el sentido de la temperatura, al cual llamaremos *parámetro de control*, y supondremos que existen una estructura de vecindad, definida en el capítulo cuatro, y un mecanismo de generación. Consideremos la siguiente definición:

Definición 25 *Sea* (Γ, k) *una instancia de un* **POC** Π, $\gamma_i, \gamma_j \in \Gamma$ *dos soluciones de* Π, *con* $k(\gamma_i)$, $k(\gamma_j)$ *sus costos respectivos, el* criterio de aceptación *determinará si* γ_j *es aceptado a partir de* γ_i *con la siguiente función de probabilidad, la cual consideraremos como la función heurística:*

$$h_c = \begin{cases} 1 & si\ k(\gamma_j) leq k(\gamma_i) \\ exp\ \frac{k(\gamma_i)-k(\gamma_j}{c}) & si\ k(\gamma_j) > k(\gamma_i) \end{cases}$$

donde $c \in \Re^+$ *denota el parámetro de control.*

Con lo anterior se puede plantear el algoritmo **rs**, más detalles en *[Aarts, Laarhoven (1988)]*, para resolver un **POC**. Sin embargo, lo que haremos en seguida es modificar el algoritmo **rs** para que funcione como un algoritmo heurístico para problemas de búsqueda.

La modificación se basa en el método primero mejor **PM**, en el cual se guardan en la lista *abiertos* todos los posibles estados encontrados por la búsqueda en el espacio de estados, y se escoge aquel que tenga la mínima evaluación heurística (calculada con el criterio de Metroplis) de la lista, lo que garantiza que en cada selección se escoge ávidamente el mejor estado, pero teniendo la posibilidad de volver a algún estado previamente encontrado, si es que el encontrado no es menor a este. Este aspecto que se adaptará a la versión usual del **rs** permite movernos de un estado a otro que tenga una mejor evaluación heurística, ya que el algortimo usual puede caer en mínimos locales.

En seguida se muestra la modificación del algoritmo **rs**. Sea v_0 el estado inicial del problema Π planteado como un problema de búsqueda sobre un espacio de estados $G_=(E_\Pi, A_\Pi)$

1. poner v_0 en la lista *PM*, $h(v_0) = 0, c_0 = 0, k := 0$;

2. si la lista de *PM* es vacía terminar con falla;

3. sacar de *PM* al estado v que cumpla con

$$h(v) = \min\{h(u) \mid u \in PM\}$$

y meter v a *cerrados*;

4. si $v \in \Gamma$ terminar con éxito;

5. determinar $V(v)$ con el mecanismo de generación
6. para todo $w \in V(v)$ calcular

$$\Delta E = h(v) - h(w)$$

   (a) si $w \notin aPM$ y $w \notin cerrados$, entonces meterlo a PM,
   (b) si $w \in PM$ o $w \in cerrados$
      i. Si $h(w) \leq h(v)$, entonces , v:=w, meter w a PM
      ii. Si $h(w) > h(v)$, entonces,
      si

$$exp\left(\frac{E}{c\,k}\right) > random[0,1), v := w$$

7. ir a 2.

Es importante hacer notar que para algunos planteamientos la función heurística h puede ser considerada como la función k del problema de optimización combinatoria.

Con esto terminamos la exposición del **PM**. En seguida veremos una especialización de éste conocido como A^*, el cual usa una función de estimación compuesta.

5.3 Estrategia A^*

A^* es una de las versiones de estrategias heurísticas de rutas más cortas, la cual fue introducida por varios investigadores: *[Michie, Ross (1970)]*, *[Slagle, Bursky (1968)]*, *[Pohl(1970)]* y *[Nilsson(1980)]*. Además, a finales de los 60's Hart, Nilsson y Rafael replantearon la estrategia para problemas de optimización cuando las estimaciones son optimistas y establecieron las propiedades básicas de la estrategia A^*.

Este procedimiento combina las funciones de evaluación vistas para la aproximación de una solución; de esta manera se tiene para cada estado v la estimación del costo del camino de él hacia un estado meta γ proporcionada por $h(v)$ y el costo que se ha acumulado hasta ese estado empezando desde el estado inicial proporcionado por la función $g(v)$. De esta manera, la función de evaluación del procedimiento A^* se definirá:

$$f(v) = g(v) + h(v)$$

donde

$$g(v_0) = 0$$
$$g(v) = g(u) + c(u, v)$$

donde $c(u, v)$ representa el costo (o dificultad) del estado u al estado v.

Observemos que la función $f(v)$ proporciona una aproximación del costo del camino del estado inicial v_0 hasta un posible estado meta γ.

Es importante notar que para el estado inicial v_0 por la construcción de la función tenemos que $f(v_0) = g(v_0) + h(v_0)$ pero como $g(v_0) = 0$ se cumple que $f(v_0) = h(v_0)$. Análogamente con el estado final γ se debe cumplir que $f(\gamma) = g(\gamma)$ porque $h(\gamma) = 0$.

El procedimiento A^* se describe a continuación; supongamos que v_0 es el estado inicial.

Sea $G_\Pi = (E_\Pi, A_\Pi)$ la gráfica que representa un espacio de estados para el probela :

1. poner v_0 en la lista *abiertos*, y hacer $f(v_0) = h(v_0)$;
2. si la lista *abiertos* es vacía terminar con falla;
3. sacar de la lista *abiertos* al estado v que cumpla con

$$f(v) = \min\{f(w) \mid w \in \textit{abiertos}\}$$

   y meter v a *cerrados*;
4. si $v \in \Gamma$ terminar con éxito;
5. Determinar $V(v)$
6. para todo $w \in V(v)$ calcular $f(w)$
   (a) Si $w \notin$ *abiertos* y $w \notin$ *cerrados*, entonces v meterlo a *abiertos*, calcular $f(w)$,
   (b) Si $w \in$ *abiertos* $w \in$ *cerrados*, comparar el nuevo valor $f(w)$ asignado a w con el valor previamente asignado, que llamamos $f'(w)$
      i. Si $f'(w) \leq f(w)$, entonces no cambiar nada,
      ii. Si $f'(w) > f(w)$, entonces, substituir $f(w)$ por $f'(w)$ y actualizar el apuntador hacia v,
7. ir a (2).

Notemos que cuando $h(v) \equiv 0$ el procedimiento A^* es semejante al procedimiento de búsqueda óptima.

Para analizar el comportamiento del método debemos considerar los siguientes conceptos:

$g^*(w) = \min\{k(v_0w\text{-}camino) \mid \forall v_0w\text{-}camino, \forall w \in E\}$

$h^*(w) = \min\{k(w\gamma\text{-}camino) \mid \forall w\gamma\text{-}camino \in G_\Pi, \gamma \in \Gamma_\Pi, \forall w \in E_\Pi\}$

$\Gamma^* = \{\gamma^* \mid \exists v_0\gamma^*\text{-}camino\}$

$f^*(v) = h^*(v) + g^*(v) = k(v_0v^*\text{-}camino) + k(v\gamma^*\text{-}camino) = k(v_0\gamma^*\text{-}camino)$. *representa el costo óptimo sobre todo los* $v_0\gamma$*-caminos* que pasan por el estado v.

Por definición se cumplen las siguientes desigualdades:

$$g^*(v) \leq g(v), \forall v \in E$$

y

$$h^*(v) \leq h(v), \forall v \in E$$

Para los estados v_0 y γ se tiene la siguiente relación:

$$f^*(v_0) = h^*(v_0) = f^*(\gamma) = g^*(\gamma)$$

Lo interesante que falta por analizar del procedimiento A^* es que es completo y además es capaz de encontrar una solución óptima.

Para poder garantizar que el procedimiento termina y además encuentra una solución óptima utilizaremos el concepto de *heurística admisible* que se define como:

Definición 26 *Sea* $G_\Pi = (E_\Pi, a_\Pi)$ *un espacio de estados y* h *una función heurística definida para ese espacio,* h *es admisible si*

$$\forall v \in E_\Pi,\ h(v) \leq h^*(v)$$

Es decir, una heurística será admisible en el estado v si es una cota inferior de la estimación mínima del estado v a γ.

Una aclaración necesaria es que algunos autores de inteligencia artificial como *[Shirai, Tsujii (1982)]*, le dan el nombre de A^* al procedimiento anterior sólo cuando este último usa una función heurística admisible.

En primera instancia una heurística admisible sería cuando $h(v) \equiv 0$ para todo v, pero eso convertiría al procedimiento de A^* en un procedimiento de fuerza bruta.

Lo que se demostrará más adelante es que si el algoritmo A^* usa heurísticas admisibles siempre encontrará la solución óptima, pero antes se demostrarán propiedades importantes del método A^*.

Lema 1 *Sea $G_\Pi = (E_\Pi, A_\Pi)$ un espacio de búsqueda para un problema Π y sea h una heurística admisible $\forall v \in E_\Pi$, entonces en algún momento antes de que termine A^* se cumple que para todo $v_0\gamma^*$-camino existe $w \in$ abiertos, $w \in v_0\gamma^*-$camino y cumple con $f(w) \leq f^*(v_0)$.*

Demostración. *Sea*

$$v_0\gamma^*-camino = v_0, v_1, v_2, v_3, \ldots, w, v_{i+1}, v_{i+2}, \ldots, \gamma^*$$

Como el procedimiento todavía no termina, entonces debe existir un vértice del $v_0\gamma^*$-camino que está en la lista de abiertos.

Sea w el vértice que está en abiertos, entonces

$$f(w) = h(w) + g(w)$$

pero notemos que

$$g^*(w) = g(w)$$

y como h es una función admisible se cumple que

$$h(w) \leq h^*(w)$$

por lo que tenemos

$$f(w) = h(w) + g(w) \leq h^*(w) + g^*(w) = f^*(w)$$

pero como

$$f^*(w) = f^*(v_0)$$

se cumple que

$$f(w) \leq f^*(v_0)$$

□

El lema anterior será de mucha utilidad para demostrar la completez y la optimalidad del algoritmo A^*. Consideremos a continuación otro resultado útil.

Lema 2 *Sea* $G_\Pi = (E_\Pi, A_\Pi)$ *un espacio de estados para un problema* Π. *Y sea además* $d(v)$ *que denota la longitud mínima con repecto al número de aristas del* v_0v*-camino , entonces*

$$g^*(v) \geq d(v) \cdot c^*(x,y)$$

donde

$$c^*(x,y) = \min\{c(w,z) \mid (w,z) \in v_0v\text{-}camino\}$$

Demostración. *Denotemos el camino de longitud mínima de* v_0 *a* v *como*

$$v_0, w_1, w_2, \ldots, w_{k-2}, w_{k-1}$$

donde

$$w_{k-1} = v$$

notemos que

$$d(v) = k$$

Sea el camino de costo mínimo de v_0 *a* v

$$v_0, v_1, v_2, \ldots, v_{n-2}, v_{n-1}$$

donde

$$v_{n-1} = v$$

es claro que la longitud de este camino es de n aristas.

Obsérvemos que se debe cumplir que $k \leq n$, *puesto que si no fuera así, es decir que* $n < k$, *entonces el camino de longitud mínima valdría n, lo que contradiría el hecho de que* $d(v)$ *fuera la longitud mínima del* $v_0v - camino$. *Además como se cumple que*

$$\forall (w,z) \in A_\Pi, c(w,z) > 0$$

se cumple la siguiente relación

$$g^*(v) = \sum_{(w,z)\in v_0v-camino} c(w,z) \geq n \geq k = d(v)$$

Denotemos

$$c_i = c(v_i, v_{i+1}), \forall i = 0, \ldots, n-2$$

entonces la siguiente propiedad se cumple

$$\sum_{i=0}^{n-2} c_i = c_0 + c_1 + c_2 + \ldots + c_{n-2} \geq \underbrace{1+1+1+\ldots+1}_{k-veces}$$

Sea $c^* = \min\{c_i\}$, *entonces tenemos que*

$$\sum_{i=0}^{n-2} c_i \geq \underbrace{c^*+c^*+c^*+\ldots+c^*}_{k-veces} = k \cdot c^* = d(v) \cdot c^*$$

$\Longleftrightarrow g^*(v) \geq d(v) \cdot c^*$ □

El siguiente resultado nos garantiza la terminación del algoritmo A^*, basándose en el concepto de heurística admisible.

Teorema 8 *Sea $G_\Pi = (E_\Pi, A_\Pi)$ un espacio de estados de un problema Π y h una heurística admisible para A^*. Si existe un $v_0\gamma$-camino$\in G$, entonces A^* siempre lo podrá encontrar.*

Demostración. *Supongamos que el algoritmo A^* no termina.*

Como el número de ramificaciones para cada vértice es finito y el proceso de búsqueda continua se tiene que

$$\forall v \in \textit{abiertos}$$

se cumple que

$$d(v) > M$$

para M suficientemente grande. Por el lema 2 tenemos que

$$g^*(v) \geq d(v) \cdot c^*$$

pero como

$$g(v) \geq g^*(v)$$

y

$$f(v) \geq g(v)$$

entonces

$$f(v) \geq d(v) \cdot c^*$$

si y sólo si

$$f(v) \geq d(v) \cdot c^* \cdot \frac{f^*(v_o)}{f^*(v_o)}$$

si y sólo si

$$f(v) \geq d(v) \cdot \frac{f^*(v_o)}{\frac{f^*(v_o)}{c^*}} = \frac{d(v)}{\frac{f^*(v_o)}{c^*}} \cdot f^*(v_o).$$

Sea

$$M = \frac{f^*(v_o)}{c^*}$$

y como

$$d(v) > M$$

se cumple que

$$\frac{d(v)}{M} > 1$$

Por lo tanto tenemos

$$f(v) \geq \frac{d(v)}{M} \cdot f^*(v_o) > f^*(v_o)$$

Lo que contradice el lema 1. La contradicción viene de suponer que A^* no termina. Por lo tanto si existe un $v_0\gamma$-camino$\in G$, A^* lo encuentra □

El siguiente teorema garantiza la optimalidad del algoritmo A^* sobre un espacio de búsqueda.

Teorema 9 *Sea $G_\Pi = (E_\Pi, A_\Pi)$ un espacio de búsqueda de un problema Π, y h es una función heurística admisible, entonces A^* encuentra una solución óptima, si es que existe.*

Demostración. *Supongamos que A^* no encuentra una solución óptima.*

Por el teorema 8, sabemos que A^* efectivamente encuentra una solución, por lo que se debe cumplir que

$$f(\gamma) = h(\gamma) + g(\gamma) = g(\gamma) > f^*(\gamma), \gamma \in \Gamma$$

Pero cuando γ fue sacado de la lista de abiertos debió de haber cumplido con

$$f(\gamma) \leq f(v), \forall v \in \textit{abiertos}$$

y como $\gamma \notin \Gamma^*$ se debe cumplir

$$f^*(\gamma) < f(\gamma) \leq f(v), \forall v \in \textit{abiertos}$$

después de la terminación de A^* se tiene que

$$f^*(\gamma) = f^*(v_0) < f(v), \forall v \in \textit{abiertos}$$

lo que es una contradicción al lema 1. La contradicción viene de suponer que A^* no encuentra la solución óptima.

Por lo tanto A^* siempre encontrará la solución óptima de Π □

Sin embargo, si observamos el procedimiento A^* podremos notar que los vértices considerados para la expansión pueden ser metidos varias veces a las listas de *cerrados* y *abiertos* respectivamente.

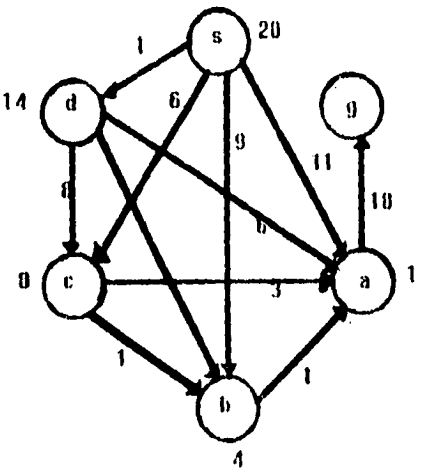


figura 5-4 Un espacio de estados

s	*a*	*b*	*c*	*d*	*g*
20	—	—	—	—	—
20*	12	13	14	15	—
20*	12*	13	14	15	29
20*	11	13*	14	15	29
20*	11*	13*	14	15	28
20*	10	11	14*	15	28
20*	10*	11	14*	15	27
20*	9	11*	14*	15	27
20*	9*	11*	14*	15	26
20*	8	9	10	15*	26
20*	8*	9	10	15*	25
20*	7	9*	10	15*	25
20*	7*	9*	10	15*	24
20*	6	7	10*	15*	24
20*	6*	7	10*	15*	23
20*	5	7*	10*	15*	23
20*	5*	7*	10*	15*	22
20*	5*	7*	10*	15*	22

figura 5-5 En la tabla se muestra el número de veces que un estado fue metido a la lista *cerrados* con *

Por ejemplo si consideremos el espacio de estados de la *figura 5-4* notaremos que el total de veces que el vértice *A* fue metido y sacado de las listas es de 17, como se muestra en la *figura 5-5* y en la *figura 5-6* , lo que implicaría que el orden de complejidad estimado para el procedimiento sería $O(2^n)$, donde n es el número de vértices del espacio de estados, ver *[Martelli(1977)]*.

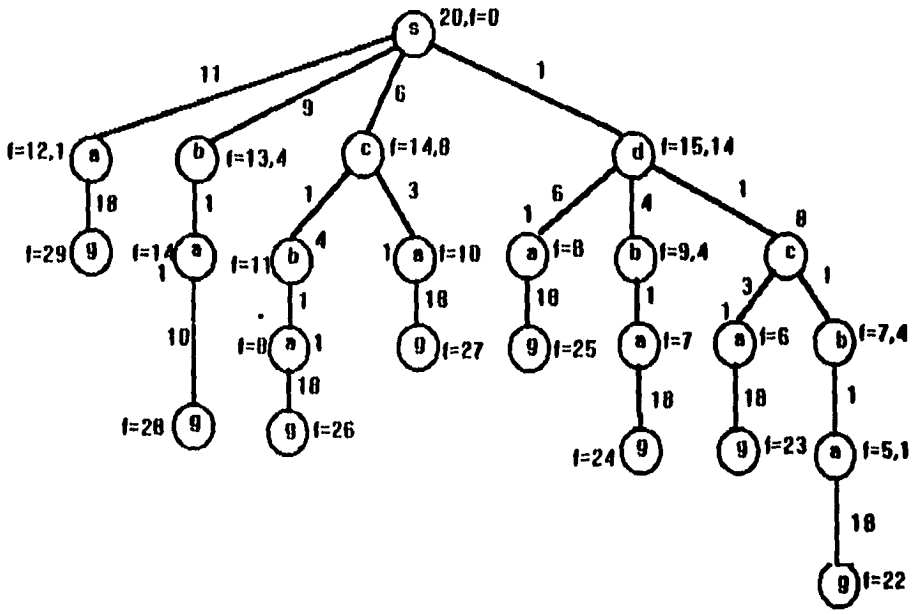


figura 5-5 Representación de la búsqueda en el espacio de estados

Es importante notar que la mayoría de los vértices del ejemplo anterior cumplen con la propiedad que

$$h(v) > h(w) + c(v, w)$$

lo cual es equivalente a

$$h(v) - h(w) > c(v, w)$$

es decir, si consideramos $h(v) - h(w) = \Delta h$ como el incremento de la estimación de w con respecto a v, entonces tendríamos que

$$\Delta h - c(v, w) > 0$$

lo que significaría que la estimación del costo $k(w\gamma - camino)$ aumentaría en al menos $\Delta h - c(v, w)$.

Pero si sucediera lo contrario para todos los vértices, es decir

$$h(v) \leq h(w) + c(v,w), \forall v \in E, \text{ y } w \in V(v)$$

entonces tendríamos que

$$\Delta h \leq c(v,w)$$

o

$$\Delta h - c(v,w) \leq 0$$

Con esta relación aumentaría la certeza en la estimación, como se demostrará más adelante; de esta manera tenemos la siguiente definición.

Definición 27 *Sea* $G_\Pi = (E_\Pi, A_\Pi)$ *un espacio de estados para un problema* Π, *y sea* h *una función heurística, se dice que* h *es* monótona *si cumple que*

$$h(v) \leq h(w) + c(v,w), \forall v \in E_\Pi, \text{ y } w \in V(v)$$

Por lo anterior, lo deseable siempre será encontrar heurísticas que sean monótonas, pues en caso contrario la complejidad del algoritmo A^* se vuelve exponencial en el peor de los casos, *[Passino, Panos (1994)]*. Desafortunadamente no es fácil especificar funciones heurísticas que sean monótonas.

Lo que se proprondrá enseguida es el uso de espacios métricos para encontar heurísticas monótonas y admisibles.

Para empezar consideraremos un conjunto arbitrario no vacío S y sea $\varphi : S \times S \rightarrow \Re$, donde φ tiene las siguientes propiedades:

(a) $\varphi(x,y) \geq 0$, para todo $x, y \in S$, y $\varphi(x,y) = 0$, si y sólo si $x = y$

(b) $\varphi(x,y) = \varphi(y,x)$ para todo $x, y \in S$

(c) $\varphi(x,y) \geq \varphi(x,z) + \varphi(z,y)$, para todo $x, y, z \in S$

Usualmente a φ se le llama *métrica* y a la pareja $\{S, \varphi\}$ se le llama *espacio métrico*. Consideremos ahora que $z \in S$ y definimos

$$d(z,S) = \inf\{\varphi(z,z') \mid z' \in S\}$$

El valor de $d(z,S)$ se le llama *la distancia entre* z y el conjunto S.

El primer resultado que se mencionará, es que si la función heurística es escogida como la distancia entre el estado v y el conjunto Γ, de manera similar a

como definimos un espacio métrico, y la métrica cumple con ciertas condiciones, entonces la heurística será admisible y monótona.

Teorema 10 *Sea* Π *un* **pb**, *y* $G = (E_\Pi, A_\Pi)$ *la gráfica que lo representa, si*

$$h(x) = \inf\{\varphi(v,\gamma) \mid \gamma \in \Gamma\}$$

y φ *es un métrica en* E_Π *con*

$$\varphi(x,y) \leq c(x,y), \text{ para toda } (x,y) \in A_\Pi$$

entonces la métrica es admisible y monótona.

Demostración. *Para probar que la heurísitca* $h(x)$ *es admisible sea*

$$x\gamma\text{-camino, donde } v_0 \in E_\Pi$$

y sean u, v *dos estados consecutivos en* $x\gamma$*-camino.*

Por la desigualdad del triángulo se tiene que:

$$\varphi(u,w) \geq \varphi(u,v) + \varphi(v,w)$$

donde v, w *son consecutivos en* $x\gamma$*-camino. Usando de manera repetida la desigualdad a lo largo del* $x\gamma$*-camino tenemos que*

$$\varphi(x,\gamma) \leq \sum_{(u,v)\in x\gamma\text{-camino}} \varphi(u,v)$$

y como por hipótesis se tiene que $\varphi(u,v) \leq c(u,v)$, *para toda* $(u,v) \in A_\Pi$, *entonces*

$$0 \leq \sum_{(u,v)\in x\gamma\text{-camino}} \varphi(u,v) \leq \sum_{(u,v)\in x\gamma\text{-camino}} c(u,v) = k(x\gamma\text{-camino})$$

y como esta desigualdad es verdadera para todos los posibles $x\gamma$*-camino's, en particular lo es para* $x\gamma^*$*-camino entonces se cumple que*

$$0 \leq \sum_{(u,v)\in x\gamma\text{-camino}} \varphi(u,v) = h(x) \leq \sum_{(u,v)\in x\gamma\text{-camino}} c(u,v) = k(x\gamma^*$$

Para demostrar que $h(x)$ *es monótona, sea* $x\gamma$*-camino de* G_Π, *sabemos que para todo* $u, v \in x\gamma$*-camino, se les asigna la evaluación* $h(v) = \inf\{\varphi(v,\gamma) \mid \gamma \in \Gamma\}$ *y* $h(w) = \inf\{\varphi(w,\gamma) \mid \gamma \in \Gamma\}$. *Donde, para cada evaluación se tiene que*

$$h(x) = \inf\{\varphi(v,\gamma) \mid \gamma \in \Gamma\} = \inf\left\{\sum_{(r,s)\in x\gamma\text{-camino}} \varphi(r,s)\right\}$$

Y como v y w, son dos estados consecutivos se tiene que

$$h(v) = \inf\left\{\sum_{(r,s)\in v\gamma\text{-camino}} \varphi(r,s)\right\}$$

$$h(v) = \inf\left\{\varphi(v,w) + \sum_{(r,s)\in w\gamma\text{-camino}} \varphi(r,s)\right\}$$

$$h(v) = \varphi(v,w) + \inf\left\{\sum_{(r,s)\in w\gamma\text{-camino}} \varphi(r,s)\right\}$$

$$h(v) = \varphi(v,w) + h(w) \leq c(v,w) + h(w).$$

Como se escogió un camino arbitrario de G_Π, compuesto por estados también arbitrarios, se tiene que $h(x)$ es una heurística monótona □

El resultado siguiente nos garantiza que no es necesario usar la noción de distancia de un espacio métrico para las heurísticas.

Teorema 11 *Sea $\Theta : E_\Pi \times E_\Pi \to \Re^+ \cup \{0\}$, suponga que*

$$\Theta(u,v) \leq c(u,v), \forall (u,v) \in A_\Pi$$

entonces para Π existe una función heurística

$$h(x) = \inf\{\Theta(x,\gamma) \mid \gamma \in \Gamma\}$$

tal que Θ no es una métrica para E_Π, que es admisible y monótona.

Demostración: *Supongamos que $\Theta(u,v) = 0$, para toda $u, v \in E_\Pi$, entonces Θ no es una métrica, pero cuando Θ es usada como función heurística se tiene que $h(x) = 0$ para toda $x \in E_\Pi$, es una función admisible y monotona* □

Consideremos ahora, las siguientes métricas sobre E_Π definidas como sigue

$$\varphi_a(u,v) = \beta\frac{\varphi(u,v)}{1 + \varphi_a(u,v)}$$

donde

$$\beta = \inf\{c(u,v) \mid (u,v) \in A_\Pi\}$$

Sea φ_b una métrica acotada en E_Π, es decir

$$\text{para toda } (u,v) \in A_\Pi \text{ existe } \lambda > 0$$

tal que

$$\varphi_b(u,v) \leq \lambda$$

Sea $\varphi_c(u,v)$ una métrica en E_Π y supongamos que

$$\varphi_c(u,v) = \alpha c(u,v)$$

para toda $(u,v) \in A_\Pi$ y para alguna $\alpha > 0$. Además, sea $\varphi_\beta(u,v)$ una métrica en E_Π tal que

$$\varphi_\beta(u,v) = \begin{cases} \beta \text{ si } u \neq v \\ 0 \text{ si } u = v \end{cases}$$

para toda $(u,v) \in A_\Pi$.

Teorema 12 *Para un* **pb** Π *las siguientes funciones heurísticas son admisibles y monótonas:*

1. $h_1(x) = \inf\{\varphi_a(x,\gamma) \mid \gamma \in \Gamma\}$
2. $h_2(x) = \inf\left\{\frac{\beta}{\lambda}\varphi_b(x,\gamma) \mid \gamma \in \Gamma\right\}$
3. $h_3(x) = \inf\left\{\frac{1}{\alpha}\varphi_c(x,\gamma) \mid \gamma \in \Gamma\right\}$
4. $h_4(x) = \inf\{\varphi_\beta(x,\gamma) \mid \gamma \in \Gamma\}$

Demostración. La demostración se sigue del *teorema 10* □

El teorema anterior nos dice cómo podemos plantear funciones heurísticas que sean admisibles y monótonas y con los resultados anteriores, cada vez que se use una de estas funciones se podrá garantizar la optimalidad del algoritmo A^*.

Consideremos ahora el ejemplo de una variante del probelam de asinación de horarios (**(ph)**) al que denominaremos problema de asignación de tareas (**pt**) .

Ejemplo 11 *Supongamos una colección de tareas, $t_1, t_2, \ldots$ con tiempos de ejecución $d_1, d_2, \ldots$ respectivamente. Las tareas serán ejecutadas por un conjunto de m procesadores idénticos. En general, todas las tareas pueden ser ejecutadas por algún procesador, pero cada procesador sólo puede ejecutar una tarea a la vez. Además, existe una relación de precedencia entre las tareas, la cual nos indica qué tarea debe ser terminada antes de que otra tarea empiece.*

En este caso se pretende asignar las tareas a los procesadores de tal forma que la relación de precedencia no sea violada y que todas las tareas sean procesadas en el menor tiempo posible. Lo que se quiere es minimizar el tiempo que lleva procesar todas las tareas.

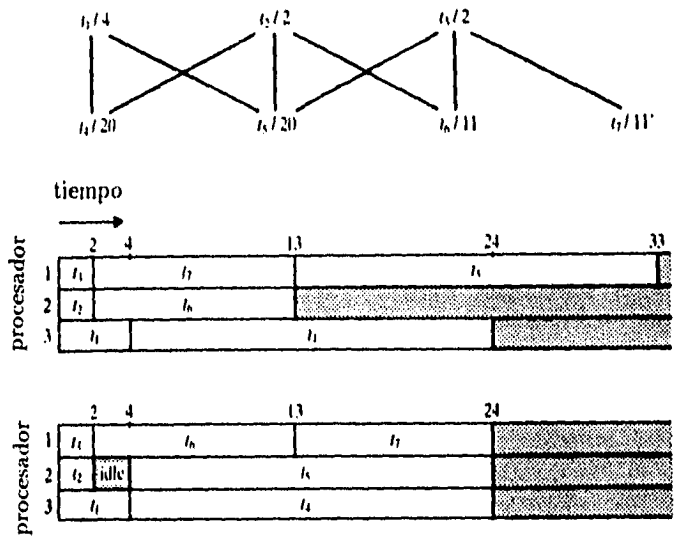


figura 5-6 Problema de asignación de siete tareas y tres procesadores; en la parte de arriba se muestra la relación de precedencia y la duración de las tareas. Por ejemplo la tarea t_5 tiene una duración de 20 unidades de tiempo, y su ejecución sólo puede empezar después de que las tareas t_1, t_2 y t_3 sean realizadas. Se muestran dos asignaciones factibles; la primera es una asignación suboptimal con un tiempo de duración de 33 unidades de tiempo,la segunda es la asignación óptima con un tiempo de término de 23 unidades de tiempo. En este problema las asiganaciones óptimas pueden tener un tarea ficticia de retraso

La manera en que construiremos un horario es la siguiente. Primero empezaremos con un horario vacío, donde cada procesador no tiene asignada ninguna tarea; gradualmente se insertarán tareas una por una en un posible horario hasta que todas las tareas sean insertadas. En general habrá alternativas en cada inserción porque habrá varios candidatos.

El planteamiento que propondremos para el **ph** *es el siguiente:*

1. *Los horarios parciales serán representados por estados.*
2. *El estado inicial es el horario vacío.*
3. *Un estado sucesor de algun horario parcial se obtiene agregando una tarea no asignada al horario; otra posibilidad es dejar libre un procesador que haya completado una tarea ficticia actual.*
4. *Cualquier horario que incluya todas las tareas en el problema es un estado final.*

5. *El costo de la solución es el tiempo total de un estado final.*

6. *El costo de una transición entre dos horarios (parciales), los cuales tienen un tiempo final f_1 y f_2 respectivamente, es la diferencia $f_2 - f_1$.*

Para hacer la representación de los horarios parciales, necesitamos la siguiente información.

- *una lista de las tareas en espera con sus tiempos de ejecución*
- *la asignacióna actual para cada procesador*

Además agregaremos, por conveniencia:

- *el tiempo que ocupa el horario (parcial), es decir el tiempo total de la asignación actual.*

Consideremos ahora la función heurística, la función que se propondrá es una heurística admisible, lo que implicará que encontraremos un horario óptimo.

La función será una estimación optimista del tiempo de finalización de un horario parcial, acompañado de las tareas en espera restantes. Esta estimación será calculada bajo la suposición que dos restricciones del horario real son relajadas:

1. *Se removeran las restricciones de precedencia*
2. *se colocará (ficticiamente) una tarea que pueda ser ejecutada de manera distribuida en varios procesadores, y que la suma de los tiempos de ejecución de esta tarea sobre todos los procesadores sea igual a el tiempo de ejecución originalmente especificado por un sólo procesador.*

Sean los tiempos de ejecución de las tareas actuales $d_1, d_2 \ldots$ y los tiempos de término de las asignaciones actuales de los procesadores son $f_1, f_2 \ldots$ tal que la estimación para las tareas en espera es

$$F = \frac{(\sum_i d_i + \sum_j F_j)}{m}$$

Donde m es el número de procesadores. Sea el tiempo final de el horario actual

$$Fin = \max_j(F_j)$$

entonces la estimación heurística h es

$$Si\ F > Fin\ entonces\ h = F - Fin\ sino\ h = 0$$

Proponemos también un programa hecho en PROLOG que resuelve el problema, ver Apéndice B.

Hemos venido mostrando las posibles estrategias que se pueden usar para la resolución de problemas provenientes, tipicamente, de la Inteligencia Artificial (**IA**), las cuales por ser de esa área del conocimiento han sido poco consideradas, por la habitual creencia de que la **IA** es una parte de la ciencias computacioneles dedicada a tratar de hacer que las máquinas realicen actividades que los humanos consideramos inteligentes. Sin embargo, los problemas existenciales de la **IA** están siendo superados y ahora se le puede considerar como una parte de las ciencias computacionales que se especializa en rosolver problemas, que por su naturaleza no tienen un procedimiento (algoritmo) que sea eficiente o que incluso ni siquiera exista (es decir que el problema sea indecidible).

Por lo anterior, el objetivo del presente trabajo fue mostrar, precesiamente, otro enfoque para la resolución de problemas en optimizació combinatoria.

6. CONCLUSIONES

Durante el trabajo mostramos que un primer paso para resolver una instancia de un problema Π es saber si el lenguaje que representa a Π bajo el esquema de codificación **e**, $L[\Pi, \mathbf{e}]$, está en **P**, **NP** o **NP-C**.

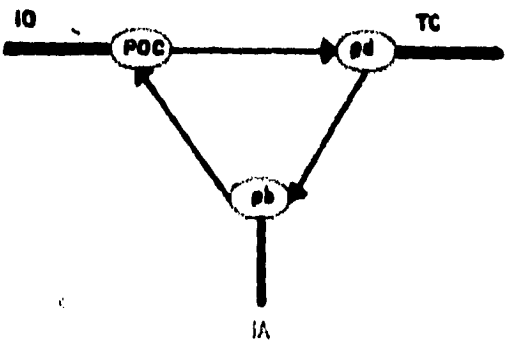
Si el lenguaje $L[\Pi, \mathbf{e}]$ está en la clase **P**, tenemos que buscar una buena estrategia para encontrar la solución óptima de Π (si somos afortunados, podremos plantear a Π como algún modelo conocido, programación lineal, problemas de redes o programación dinámica entre otros).

Si el lenguaje $L[\Pi, \mathbf{e}]$ está en **NP** o **NP-C**, sabemos que existe un algoritmo en tiempo exponencial que resuelva a Π, y además como hemos supuesto que $\mathbf{NP} \neq \mathbf{P}$, entonces el problema Π no puede ser resuelto en tiempo polinomial. Sin embargo, como queremos resolver Π, en un tiempo razonable, debemos pensar en posibles alternativas: el Recocido Simulado, la Búsqueda Tabú, Algoritmos Genéticos o Algoritmos Glotones, entre otros, pues estas son opciones que, en general, han demostrado un buen desempeño en Investigación de Operaciones.

Los métodos provenientes de la Inteligencia Artificial, presentados en este trabajo, tienen la característica de poder representar al problema Π en un espacio de estados y por medio de su exploración, poder encontrar una solución. La búsqueda en los espacios de estados se guía por la función heurística h, la cual, al tener las características de ser admisible y monótona, garantizará que el algoritmo de búsqueda heurístico (**PM** o A^*) tendrá un buen desempeño y además encontrará una solución, si es que existe.

1. Algunas reflexiones finales

Un aspecto importante de los tipos de problemas que se estudiaron es que dado inicialmente un Problema de Optimización Combinatoria Π, a este lo podemos plantear como un Problema de Decisión, $\mathbf{pd}_\Pi$, similarmente a $\mathbf{pd}_\Pi$ lo podemos plantear como un Problema de Búsqueda $\mathbf{pb}_\Pi$, y finalmete, a $\mathbf{pb}_\Pi$ lo podemos plantear como un problema de optimización combinatoria, es decir el siguiente diagrama conmuta



En el diagrama anterior se muestra la relación de los tipos de problemas estudiados y el área de conocimiento a la que pertenecen:

- IO: Investigación de Operaciones,
- IA: Inteligencia Artificial,
- TC: Teoría de la Computación)

Consideremos como referencia inicial a los problemas de de decisión. La versión de Π como problema de decisión necesita de un lenguaje y un esquema de codificación $\mathbf{e}$. Si $L[\Pi, \mathbf{e}]$ está contenido en alguna clase arbitraria de problemas $\mathbf{\Pi} \in \{\mathbf{P}, \mathbf{NP}, \mathbf{NP-C}\}$, la versión de Π como problema de búsqueda también estará en $\mathbf{\Pi}$, análogamente, la versión del problema de búsqueda planteado como problema de optimización combinatoria estará tambien en Π y finalmente, el problema de optimización combinatoria expresado como un problema de decisión estará en $\mathbf{\Pi}$, lo que implica que la complejidad de Π se preserva, sin importar la forma en que sea planteado.

Supongamos nuevamente, que queremos resolver una isntancia de $\Pi \in \mathbf{NP}$, también supongamos por un momento que somos capaces de construir unalgoritmo heurístico H (ya sea en el sentido de la investigación de Operaciones o en el de Inteligencia Artificial) que sea lo sufientemente potente para resolver cualquier instancia de Π en tiempo polinomial, y con base en la característica del parrafo anterior para las diferentes clases de problemas, podemos formular la siguiente proposición:

Sea $\Pi \in \mathbf{NP}$ y H un algortimo como se especifico anteriormente, si H resuleve a Π en tiempo polinomial, entonces $\mathbf{P} = \mathbf{NP}$

La demostración de la proposición anterior se lograría por la existencia de H y de nuestra capacidad para encontrarla; sin embargo, la búsqueda de H finalizará si se logra demostrar que $\mathbf{P} \neq \mathbf{NP}$.

7. APÉNDICE A

7.1 Tabla de Símbolos

Π Un problema
(POC) Problemas de Optimización Combinatoria
(pd) problema de decisión
(pb) problema de búsqueda
(Γ, k) Instancia de un POC
Γ Es el conjunto de soluciones de un POC
$\gamma^* \in \Gamma$ solución óptima de un POC
γ solución de un POC
$k : \Gamma \rightarrow \Re$ función de costo
$\Re$ El conjunto de los números reales
$\Re^+$ El conjunto de los números reales positivos
N el conjunto de los números naturales
Z el conjunto de los números enteros
$\mathbf{Z}^+$ el conjunto de los números enteros positivos
$| C |$ Cardinalidad del conjunto C
Σ en alfabeto
Σ^* la cerradura de Kleene
$| x |$ longitud de una la cadena x
ε
$L \subseteq \Sigma^*$
D_Π Conjunto de instancias del problema Π
$Y_\Pi \subset D_\Pi$ instancias cuya respuesta es *sí* para el problema Π
e esquema de codificación
$L[\Pi, \mathbf{e}]$ lenguaje asociado con el pro-blema Π y el esquema de codificación **e**,
MT *máquina de Turing*
MTD *máquina de Turing determinística*
M Una **MTD**
MTND *máquina de Turing no determinística*
N Una **MTND**
q_Y es el *estado de aceptación* en una MT
q_N es el *estado de rechazo* en una MT
$\mathbf{O}(s(n))$ se le conoce como el *Orden de complejidad s*
$p(n)$ polinomio p
P clase de problemas *polinomiales*
NP clase de problemas *no determinísticamente polinomiales*

NP-C clase de problemas *no determinísticamente polinomiales completos*

7.2 Gráficas

Se ha mencionado acerca de la Teoría de las Gráficas sin establecer una especificación, por lo que con este precedente, es necesario considerar la siguiente definición de *[Harary (1971)]*.

Definición 28 *Una gráfica* $G = (V_G, A_G)$ *consiste de un conjunto no vacío* V_G *cuyos elementos se llaman* vértices *y de un conjunto* A_G *de pares no ordenados de elementos distintos de* V_G. *Cada pareja en* A_G *es llamada* arista.

Sin embargo, también existen otros tipos de estructuras donde el orden de las parejas (aristas) es válido. Ese tipo de estructuras se les conoce como *Digráficas* ó *Gráficas Dirigidas*. Formalmente diremos:

Definición 29 *Una digráfica* $D = (N_D, E_D)$, *donde* N_D *es un conjunto de* nodos *no vacío y* $E_D \subseteq N_D \times N_D$; *los elementos de* E_D *son denominados* arcos.

Con la caracterización dada de arcos, podemos decir que para toda pareja (u, v), u es el vértice inicial y v es el vértice final.

También diremos que dos vértices (nodos) son adyacentes si existe una arista (arco) que los une. Así mismo, se aceptará que los vértices v y w son incidentes a la arista (v, w).

En ocasiones, algún tipo de problemas sólo requerirá de una "parte" de una gráfica o digráfica y entonces podemos proponer:

Definición 30 *Sea* $G = (V_G, A_G)$ *una gráfica,* $G_p(V_p, A_p)$ *es una* gráfica parcial *si,* $V_p \subseteq V_G$ *y* $A_p \subseteq A_G$.

Definición 31 *Sea* $G = (V_G, A_G)$ *una gráfica,* $G_S = (V_S, A_S)$ *es una* subgráfica *si* , $V_S \subseteq V$ *y* $A_{S=}\{(v, w) \mid v, w \in V_S\}$.

También son necesarios los siguientes conceptos:

Definición 32 *Sea $G = (V_G, A_G)$ una gráfica y sea $I : V_G \to \aleph$ una función biyectiva que a cada vértice le asocia una y sólo una etiqueta.*

Una sucesión de la forma:

$$v_i v_n\text{-camino} = v_i(v_i, v_j)v_j(v_j, v_k)v_k \cdots v_m(v_m, v_n)v_n$$

o simplemente

$$v_i v_n\text{-camino} = v_i, v_j, v_k, \ldots, v_m, v_n$$

es un camino de v_i a v_n en la gráfica G.

Si $v_i = v_n$, entonces se dirá que es un ciclo, y diremos que el $v_i v_n$it -camino es elemental si ningún vértice del camino aparece más de una vez en la sucesión

Definición 33 *Sea $G = (V_G, A_G)$ una gráfica, G es una gráfica conexa si para todo par de vértices existe un camino que los une.*

Notése que estas definiciones también son válidas para digráficas. Una consideración importante es que, dada una gráfica $G = (V_G, A_G)$; ¿Cómo puede ser representada en una forma conveniente, de tal manera que no ocupe demasiado espacio?.

Para este problema se pueden encontrar dos tipos de representaciones usuales, que se verán a continuación, y ambas suponen que:

$$G = (V, A) \text{ con } |V| = n, |A| = m \text{ y } V = \{v_1, v_2, v_3, \ldots, v_n\}$$

1. Matrices de Adyacencia. G puede ser representada por una matriz $E_{n\times n}$ donde $E(a_{ij})$ está definida como:

$$a_{ij} = \begin{cases} 1 \text{ si } (v_i, v_j) \in A_G \\ 0 \text{ en otro caso} \end{cases}$$

Esta representación es conveniente para algoritmos en gráficas, los cuales frecuentemente necesitan conocimiento de las aristas que están presentes.

Pero la principal desventaja es que requiere de espacio de $O(n^2)$, a pesar de que m sea un número mucho menor que n, lo que significaría que el número de 1's en la matriz sería menor al de 0's.

Aunado a esto, la inicialización de la matriz de manera directa requiere de $O(n^2)$ operaciones, lo que superaría a algoritmos cuya complejidad fuera menor a esta.

2. Listas de Adyacencia. La otra posible representación para una gráfica está dada por listas. En ésta representación cada vértice v tiene una lista de todos los vértices w adyacentes a él. Por lo que una gráfica puede ser representada por n listas de adyacencias, una por cada vértice, ver *figura 7-1*. Note que el espacio que ocupa una lista de adyacencia es proporcional a $n + m$. En general la lista de adyacencia es muy usada cuando $m << n$.

Es necesario observar que cuando se consideran gráficas la matriz de adyacencia tiene la propiedad de ser simétrica (situación que no se presenta con digraficas, ver *figura 7-1*), mientras que en las listas de adyacencias, cada $(v, w) \in A_G$ es representada dos veces, una vez en la lista de v y la otra en la lista de w.

Un concepto importante que debe aclararse es cuando se dice que una gráfica puede representar un espacio de estados para un problema dado (no necesa-riamente de decisión) si el problema se puede representar en términos de un vértice (nodo) inicial y un vértice (nodo) meta.

Esto significa que este problema puede ser construido como un problema de búsqueda a través de los vértices.

vértice	*adyacencias*
r	$\bullet \longmapsto s\bullet \longmapsto v\bullet \longmapsto nil$
s	$\bullet \longmapsto r\bullet \longmapsto w\bullet \longmapsto nil$
t	$\bullet \longmapsto x\bullet \longmapsto u\bullet \longmapsto w\bullet \longmapsto nil$
u	$\bullet \longmapsto t\bullet \longmapsto y\bullet \longmapsto nil$
v	$\bullet \longmapsto r\bullet \longmapsto nil$
w	$\bullet \longmapsto s\bullet \longmapsto t\bullet \longmapsto x\bullet \longmapsto nil$
x	$\bullet \longmapsto t\bullet \longmapsto w\bullet \longmapsto y\bullet \longmapsto nil$
y	$\bullet \longmapsto u\bullet \longmapsto x\bullet \longmapsto nil$

(b)

$r \quad s \quad t \quad u \quad v \quad w \quad x \quad y$

$$
\begin{array}{c}
r \\ s \\ t \\ u \\ v \\ w \\ x \\ y
\end{array}
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{pmatrix}
\qquad \text{(c)}
$$

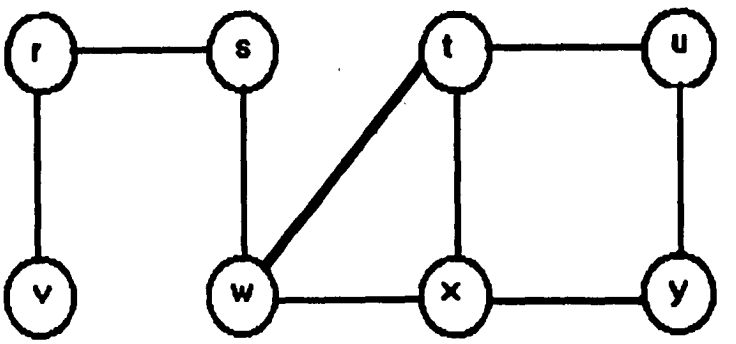


figura 7-1 Una gráfica dirigida con sus respectivas representaciones. (a) Gráfica no dirigida, (b) Matriz de adyacencia, (c) Lista de adyacencias

7.3 Búsqueda en Gráficas

La solución de problemas prácticos con técnicas de teoría de gráficas consideran el análisis en gráficas subyacentes para probar alguna propiedad. En el análisis de una gráfica G es necesario en ocasiones examinar todas las aristas (o arcos). Las dos maneras usuales para esto son:

(a) Para todo $v \in V_G$, se examinan todas las aristas incidentes a v, y se pasa a otro vértice de G el cual es adyacente a v.

(b) Para todo $v \in V_G$ se examina una sola arista incidente a v y se pasa a otro vértice de G, el cual es incidente a v.

El primer método se conoce como *Búsqueda Primero en Amplitud* (Breadth-first search, **BA**), el segundo es *Búsqueda Primero en Profundidad* (Depth-first search **BP**).

7.3.1 Búsqueda Primero en Amplitud (BA)

El método de BA es usado generalmente para identificar conexidad, ciclos y caminos más cortos con respecto al número de aristas. En general el proceso de examinación de las aristas puede ser representado por un árbol de búsqueda T, donde cada vértice del árbol es un vértice de la gráfica que será analizado.

En el inicio, un vértice v_0 de la gráfica es elegido, representando a su vez el vértice inicial de T y el *nivel* 0 del árbol. Cada una de sus aristas incidentes son examinadas, una a la vez, y son representadas por aristas incidentes a v en T. Luego, los nuevos vértices incidentes a esta arista representan el primer *nivel* de T.

Para la implantación del BA se necesitan las siguientes estructuras de datos:

(a) Lista de adyacencia para cada vértice $ady[v]$.

(b) $color[v]$: $(negro, gris, blanco)$. Al vértice v le asigna un color, inicialmente es vacío.

(c) $\pi[u]$: padre de u (es decir, el vértice que fue explorado antes que u) en T ó *nil*

(d) $d[u]$: la distancia de v_0 a u, con respecto al número de aristas.

(e) una cola Q de vértices grises, para la cual se tienen la funciones: $mete(v, Q)$ y $saca(v, Q)$, las cuales meten elementos a la cola y extraen elementos de la cola, respectivamente.También tenemos la función $frente[Q]$ que indica cual es el elemento de Q que está al frente de la cola.

ALGORITMO BA
Entrada: Una gráfica y un vértice inicial v_0, (G, v_0)
Salida: Las distancias de v_0 a todos los vértices y T.

01. **for** $v \in V$ **do**

02. $color[v] := blanco$

03. $d[v] := 0$

04. $\pi[v] := nil$

05. $color[v_0] := gris$

06. $d[v_0] := 0$

07. $\pi[v_0] := nil$

08. $mete(v_0, Q)$

09. **while** $Q \neq \emptyset$ **do**

    10. $u := frente[Q]$
    11. **for** $v \in ady[u]$ **do**
        12. **if** $color[v] = blanco$ **then**
        13. $color[v] = gris$
        14. $d[v] = d[v] + 1$
        15. $\pi[v] = u$
        16. $mete[Q, v]$
    17. $saca[u, Q]$
    18. $color[u] := negro$

En la *figura 7-1* se muestra un ejemplo de la ejecución de **BA**.

Complejidad

La inicialización es de $O(n)$. Los vértices se pintan de gris a lo más una sóla vez, por lo que las líneas (13) y (16) aseguran que cada vértice se mete una sola vez a la cola. Y las operaciones de meter y sacar toman $O(1)$.

El tiempo total de manejo de la cola es $O(n)$.

La lista de adyacencia de un vértice se revisa una sola vez antes de sacarlo, esto es por el paso (11), entonces el tiempo total en una lista de adyacencia es $O(m)$.

Por lo tanto la complejidad total es $O(n + m)$.

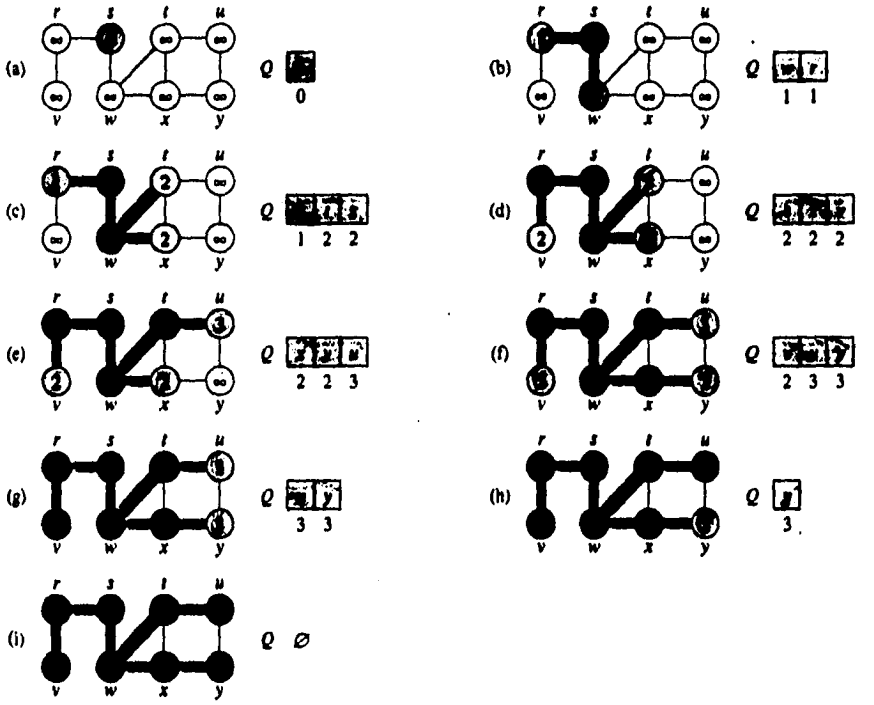


figura 7-2 Operaciones del BA sobre una gráfica

Verificación de que el BA es Correcto

Para demostrar el funcionamiento del algoritmo,demostraremos que despues de que termine el algoritmo, tendremos las distancias más cortas entre un vértice inicial v_0 y los vertices w para los cuales exista un v_0w-*camino*.

Consideremos la función

$$\delta : V \times V \to \mathbb{N}$$

tal que si $v, u \in V$, $\delta(v, u)$ es el número mínimo de aristas de un camino entre v y u.

Si no existe un camino la función toma un valor suficientemente grande.

Lema 3 $\forall(u,v) \in A$,

$$\delta(v_0, v) \leq \delta(v_0, u) + 1$$

Demostración: Supongamos que

$$\delta(v_0, v) > \delta(v_0, u) + 1$$

Además como

$$(u,v) \in A \text{ y } \delta(u,v) = 1$$

entonces

$$\delta(v_0, v) > \delta(v_0, u) + \delta(u, v)$$

lo cual es una contradicción.

Por lo tanto

$$\delta(v_0, v) \leq \delta(v_0, u) + 1 \ \square$$

Lema 4 *Sea G una gráfica sobre la cual se ha corrido BA a partir de* v_0. *Al terminar*

$$d[v] \geq \delta(v_0, v)$$

Demostración: Por inducción sobre el número de vértices que se han metido a la cola Q.

Base: Para v_0

$$\delta(v_0, v_0) = 0 = d[v_0]$$

para todos los demás

$$d[v] \geq \delta(v_0, v)$$

Inducción: Consideremos un vértice v descubierto desde u, por la asignación del paso (14) y el lema 1 tenemos

$$d[v] = d[u] + 1 \geq \delta(v_0, u) + 1 \geq \delta(v_0, v) \ \square$$

Lema 5 *Supongamos que durante la ejecución de BA la cola*

$$Q = < v_1, v_2, v_3, \ldots, v_k >$$

entonces

i. $d[v_{k+1}] \leq d[v_k] + 1$
ii. $d[v_i] \leq d[v_{i+1}]$

Demostración: Por inducción sobre el número de operaciones de la cola Q.

Base: Al inicio sólo s está en Q.

Inducción: Al sacar elementos de la cola no hay problema.

Se inserta un nuevo elemento en Q cuando se esta examinando v_k, el cual es v_k . Por el paso (16)

$$d[v_{k+1}] = d[v_k] + 1$$

con lo que se cumple (I).

Además

$$d[v_k] \leq d[v_k] + 1$$

y por hipótesis de inducción

$$d[v_k] \leq d[v_k] + 1 = d[v_{k+1}]$$

lo que prueba la parte (II) □

Teorema 13 *Supongase que se ha ejecutado* **BA** *sobre una gráfica G a partir de s, entonces*

i. Durante su ejecución **BA** alcanza todos los vértices w para los que existe un camino de v_0 a w
ii. Al terminar $d[v] = \delta(s, v)$;
iii. $\forall v \in V, v \neq v_0$, un camino más corto desde v_0 a v es el camino más corto desde v_0 a $\pi[v]$ seguido de la arista $(\pi[v], v)$.

Demostración: Sea el conjunto

$$V_k = \{v \in V \mid \delta(s, v) = k\}$$

Por hipótesis tenemos que para cada $v \in V_k$ existe un sólo momento en el cual:

(a) v es pintado de gris;

(b) a $d[v]$ se le asigna el valor k;

(c) si $v \neq s$, entonces $\pi[v] = u$ tal que $u \in V_{k-1}$.

Por inducción sobre k

Base: $k = 0$, $V_0 = \{s\}$

Inducción: Consideremos $v \in V_k$. Si v se mete a Q, entonces se mete después de todos los vértices en V_{k-1}, además todos los vértices de V_k están dentro de la cola. Esto es porque por el lema 2 los vértices

de V_k tienen que entrar después que los de V_{k-1} ya que por el *lema 3* sabemos que $d[v] = k$.

Debido a que $v \in V_k$, existe $u \in V_{k-1}$, con $(u, v) \in A$. Sea u el primer vértice pintado de gris en V_{k-1} con $(u, v) \in A$, entonces u debe ser metido a la cola. Por hipótesis de inducción, eventualmente u debe aparecer al frente de la cola. Y en el momento en que llegue al frente de la cola por el paso (11) se revisarán todas sus aristas incidentes, y se seguirá el siguiente orden:

i. En la línea (13) se pinta v de gris;
ii. En la línea (14) se hace $d[v] = d[u] + 1 = k$;
iii. En la línea (15) se hace $\pi[v] = u \in V_{k-1}$;
iv. En la línea (16) se mete v en Q.

Para concluir; si $v \in V_k$, $\pi[v] \in V_{k-1}$, entonces podemos encontrar un camino más corto de s a v utilizando las aristas de $\pi[v]$ a v □

Con esto se demuestra que el algoritmo es correcto.

Arboles de Expansión de BA

Como se dijo al iniciar este apartado, **BA** genera un árbol, el cual se puede caracterizar a partir de una gráfica $G = (V_G, A_G)$ de la siguiente manera:

$$\text{Sea } T_\pi = (V_G, A_\pi)$$

donde

$$A_\pi = \{(\pi[v], v) \in A_G \mid v \in V_G - \{v_0\}\}$$

y las aristas de A_π son llamadas aristas de árbol.

La gráfica G_π es un árbol **BA**, si V_π consiste de todos los vértices alcanzables desde v_0. Y para todo $v \in V_\pi$ existe un único camino elemental de v_0 a v en G, el cual además es el más corto (con respecto al número de aristas) en G.

La estructura del árbol **BA** tiene características bien detalladas con respecto a las aristas que no pertenecen al árbol, como son:

(a) $\forall (u, v) \in A_\pi$, $d[v] = d[u] + 1$.

(b) $\forall (u, v) \in A_G$, que van de una rama a otra en T_π (aristas de cruce) está en el mismo nivel $d[v] = d[u]$ o cumplen con $d[v] = d[u] + 1$.

7.3.2 Búsqueda Primero en Profundidad (BP)

En la estrategia de búsqueda en profundidad, las aristas de una gráfica son exploradas a partir del último vértice v que se haya encontrado en la búsqueda, de esta manera, cuando todas las aristas incidentes a v han sido exploradas la búsqueda regresa por el vértice a partir del cual se encontro a v. Este procedimiento continua hasta que se han descubierto todos los vértices que sean alcanzables a partir de algún vértice inicial. Si existen vértices que no fueran alcanzables entonces, uno de ellos se escoge y a partir de este se continua con la búsqueda, hasta que todos los vértices de la gráfica sean explorados.

Al igual que el **BA**, siempre que algún vértice v ha sido descubierto durante la exploración de la lista de adyacencia de otro vértice u, el algoritmo registra con el apuntador $\pi[v] = u$, lo cual indica también, que u es preedecesor de v. Además con el **BP** la subgráfica de precedencias (es decir el árbol generado por el algoritmo) no es único como el de **BA**, porque la búsqueda, después de haber sido empezada, en el **BP** puede iniciarse a partir de varios vértices.

La subgráfica de precedencias $T = (V_G, A_\pi)$ se caracteríza de la siguiente manera:

Sea $G = (V_G, A_G)$ una gráfica (o incluso una digráfica), entonces la aristas de T se especifican como:

$$A_\pi = \{(\pi[v], v) \mid v \in V_G, \pi[v] \neq nil\}$$

donde las aristas de A_π son llamdas aristas de árbol.

Como en **BA** los vértices son coloreados durante la búsqueda para indicar sus estados . Cada vértice inicialmente es blanco, se pinta de gris cuando es descubierto en la búsqueda y se pinta de negro cuando es abandonado, es decir, cuando su lista de adyacencia se termina de explorar.

Cada vértice tendrá dos etiquetas, la primera $d[v]$ registra cuando v fue descubierto por primera vez (y fue pintado de gris) y la segunda $f[v]$ registra cuando se termina de examinar la lista de adyacencia de v (y fue pintado de negro).

Además para todos los vértice se tien que

$$d[v] < f[v]$$

porque el vértice v es blanco antes del momento $d[v]$, es gris entre los instantes $d[v]$ y $f[v]$, y finalmente es negro.

Para la implantación del **BP** se necesitan las siguientes estructuras de datos:

(a) Lista de adyacencia para cada vértice $ady[v]$.

(b) *color*[*v*]: Se usa para determinar si un vértice fue ó no visitado (sus elementos son blanco, gris y negro).

ALGORITMO BP

Entrada: Una gráfica G (o digráfica, la cual puede ser conexa o no, en caso de que no lo sea se hablará de bosques)

Salida : Una subgráfica de precedencias

```
procedure BP-busca[u]
01. color[u] := gris
02. d[u] := t + 1
03. for v ∈ ady[u] do
        04. if color[v] = blanco then
                05. π[v] := u
                06. BP-busca[v]
07. color[u] := negro
08. f[v] := t + 1
```

```
BP(G)
01. for u ∈ V do
        02. color[u] := blanco
        03. π[u] := nil
04. t := 0
05. for u ∈ V do
        06. if color[u] = blanco then
                07. BP-busca[u]
```

La *figura 7-4* muestra el progreso del algortimo **BP** para la gráfica de la *figura 7-3.*

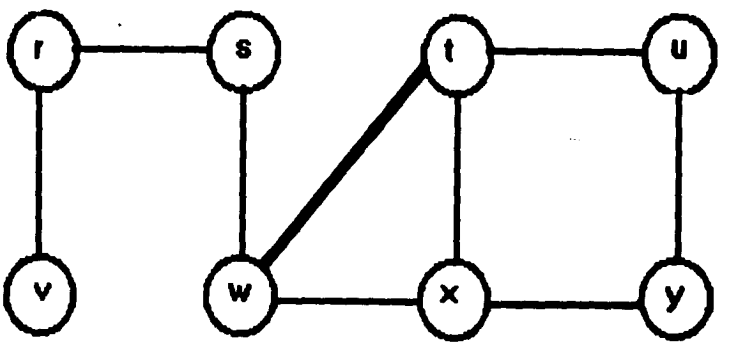


figura 7-3

Complejidad

Las instrucciones de las líneas 1-2 y 5-7 del **BP** se realizan para cada vértice, entonces tenemos un tiempo de $| V_G | = n$, que además también es el número de llamadas al procedimiento BP-*busca*. El procedimiento BP-*busca* es llamado exactamente una vez para cada vértice $v \in V_G$, BP-*busca* se invoca solamente en los vértices blancos y lo primero que hace es pintar al vértice de gris. Durante la ejecución de BP-*busca*(v), los pasos 3-6 son ejecutados $| ady(v) |$ veces, como

$$\sum_{v \in V} | ady[v] | = | A_G | = m$$

entonces el número total de ejecuciones de las líneas 2-5 en BP-*busca* es m.

Por lo que el tiempo total que ocupa el **BP** es de $m + n$.

Verificación de que BP es Correcto

En las líneas 1-3 se pintan todos los vértices de blanco y se inicializa el apuntador π con *nil*. En la línea 4 se inicializa la variable global t . Las líneas 5-7 revisan cad vértice de V y cuando encuentran un vértice blanco el algoritmo lo visita usando el procedimiento BP-*busca*. Cada vez que se llama a BP-*busca*(v) por la línea 7, el vértice u se convierte en la raíz de un nuevo árbol en la arborescencia BP.

Cuando el algortimo **BP** termina de explorar a un vértice v tiene las etiquetas $d[v]$ y $f[v]$.

En cada llamada de BP-*busca*(u), el vértice u inicialmente es blanco, y por la línea 1 se pinta a u de gris; la línea 2 registra el momento de su descubrimiento incrementando y fijando la variable t. Las líneas 3-6 examinan cada vértice v adyacente a u y recursivamente visitar a v si este es blanco. Como cada vértice $v \in ady[u]$ es considerado por las línea 3, decimos que la arista (u, v) ha sido explorada por BP, finalmente, después de que todas las aristas incidentes a u han sido exploradas, las líneas 7-8 pintan a u de negro y registran el tiempo de abandono con $f[u]$.

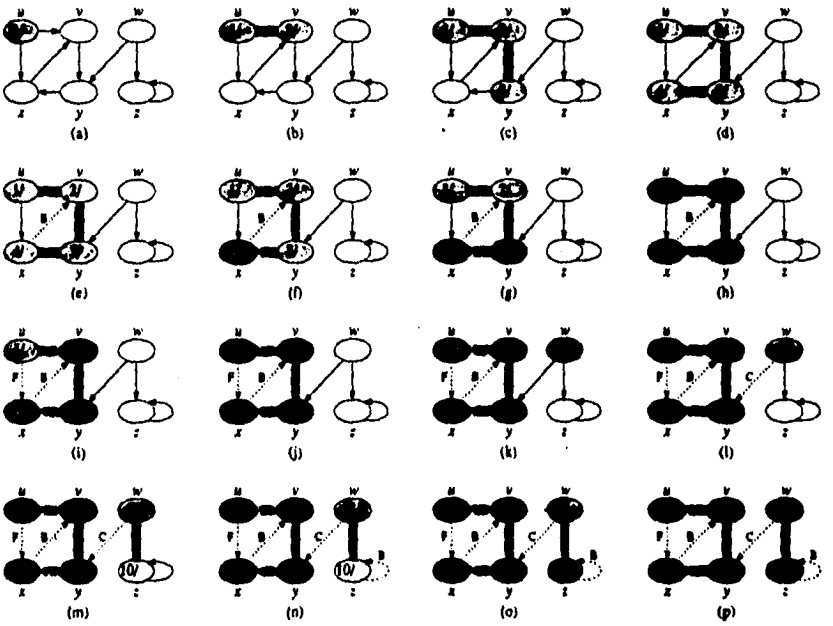


figura 7-4 Operaciones del BP sobre la gráfica de la *figura 7-3*

8. APÉNDICE B

Programa que resuelve el problema de programación de tareas por medio del algoritmo A^*

```
/* El algortimo A* esta inicializado por el predicado a-star(Start, Solution)
donde Solution es el camino del estado inicial Start al estado final goal */

a-star(Start, Sol) :-
    expand([], l(sart, 0/0), 999999, _, yes, Sol).      /* suponemos que
999999 es más grande que cualquier valor de f */

/* expand(Path, Tree, Bound, Tree1, Solved, Solution)
Path es el camino entre el estado inicial de la búsqueda y el subarbo Tree.
Tree1 es el árbol que se expande por ser el más cercano a la cota Bound. Si
el estado meta se alcanza, Solution es el camino que se encuentra del estado
inicial hasta ese estado y Solved = yes. */

/* caso 1: el estado meta es una hoja del espacio de búsqueda */

expand(P, l(N, _), _, _, yes, [N| P]) : -goal(N).

/* caso2: el valor de f es menor que la cota generamos sus sucesores */

expand(P, l(N, F/G), Bound, Tree1, Solved, S) :-
    F <= Bound,
    ( bagof( M/C, ( s(N, M, C), not member(M, P)), Succ),
    !,      /* em el estado N tiene sucesores */
    succlist( G, Succ, Ts),      /* se construyen los árbarboles Ts */
    bestf( Ts, F1),      /* el valor de f es del mejor sucesor */
    expand( P, t(N, F1/G, Ts), Bound, Tree1, Solved, S)
    ;      /* el estado N no tiene sucesores */
    Solved = never).

/* caso 3: los estados no son hojas en el espacio de estados,
se expande el estado mas promisorio */

expand(P, t(N, F/G, [T1 | Ts]), Bound, Tree1, Solved, S) :-
    F=< Bound,
    bestf( Ts, BF), min( Bound, BF, Bound1),
    expand([N | P], T, Bound1, T1, Solved1, S),
```

```
        continue( P, t(N, F/G, [T1 | Ts]), Bound, Tree1, Solved1, Solved, S).

    /* caso 4: los estados interiores del espacio no son configuraciones del problema, estos estados son podados y nunca se expandiran */

    expand(_, t(_, _, []), _, _, never, _) :- !.

    /* caso 5: el valor de f es más grande que la cota */

    expand(_, Tree, Bound, Tree, no, _) :-
        f( Tree, F), F ¿ Bound.

    /* continue(Path, Tree, Bound, NewTree, SubtreeSolved, TreeSolved, Solution) */
    continue(_, _, _, _, yes, yes, S).

    continue(P, t(N, F/G, [T1 | Ts]), Bound, Tree1, no, Solved,S) :-
        inserta(T1, Ts, NTs),
        bestf(Ts, F1),
        expand(P, t(N, F/G, Ts), Bound, Tree1, Solved, S).

    continue(P, t(N, F/G, [_ | Ts]), Bound, Tree1, never, Solved, S) :-
        bestf(Ts, F1),
        expand(P, t(N, F1/G, Ts), Bound, Tree1, never, Solved, S).

    /* succlist( G0, [Node1/Ccost,...], [l(BestNode, BestF/G),...]): se construte una lista ordenada con respecto al valor de la función f de los estados encontrandos */
    succlist(_, [], []).
    succlist( G0, [N/C | NCs], Ts) :-
        G is G0 + C,
        h(N, H),        /* el valor de la función heurística para N */
        succlist(G0, NCs, Ts1),
        inserta(l(N, F/G), Ts, Ts1).

    /* se inserta T en la lista de árboles Ts preservando el orden con respecto al valor de la función f */
    inserta(T, Ts, [T | Ts1]) :-
        f(T, F), best(Ts, F1),
        F =< F1, !.

    inserta(T, [T1 | Ts], [T1 | Ts1]) :- inserta(T, Ts, Ts1).

    /* calculo del valor de la función f */
    f( l(_, F/_), F).           /* valor de f en un estado hoja */
```

```
f( t(_, F/_, _), F).            /* el valor de f para un árbol */

bestf([T | _], F) :-
    f( T, F).       /* mejor valor de f para la lista de árboles */

bestf([], 999999).      /* no existen árboles */

min(X, Y, X) :- X=<Y, !.
min(X, Y, Y).
```

/* la siguiente parte muestra la especificación del espacio de estados para el problema de programación de tareas, la cual tiene en la base de datos la instancia particular del ejemplo 13 de la sección 5.3.

Los estados en el espacio de estados son asignaciones de actividades parciales especificadas de la siguiente manera:

[WaitingTask1/D1, WaitingTask2/D2,...] * [Task1/F1, Task2/F2,...] * FinTime

La primera lista especifica las tareas en espera y sus duraciones, la segunda lista especifica las tareas que están siendo actualmente realizadas así como sus tiempos de duración y las tareas son oredenadas de la siguiente manera: $F1 \leq F2 \leq F3 \leq ...$ *el tiempo final* es el menor tiempo de duración de la actual asignación de tareas en los procesadores */

```
/* (Nodo, FinallNodo, Cost) */

s( Task1 * [_/F | Active] * Fin1, Task2 * Active2 * Fin2, Cost) :-
    del(Task/D, Task1, Task2),      /* tomamos una tarea en espera */
    not ( member( T/_, Task2), before( T, Task) ),      /* verificación de
precedencias */
    not ( member( T1/F1, Active1),
        F < F1,
before( T1, Task) ),      /* activación de una tarea */
    Time is F + D,
    insert( Task/Time, Active1, Active2, Fin1, Fin2),      /* tiempo de
de ejcución de una tarea activada */
    Cost is Fin2 - Fin1.

/* inserción de una tarea ficticia */
s( Tasks * [_/F | Active1] * Fin, Task * Active2 * Fin, 0) :-
    insertidle( F, Active1, Active2).
    /* abandonamos una proceso inactivo */

before( T1, T2):-
    prec(T1, T2).       /* la tarea T1 precede a la tarea T2 */

before(T1, T2) :-
```

```
        prec(T, T2),
        before(T1, T).

    insert( S/A, [T/B | L], [S/A, T/B | L], F, F) :- A =< B, !.
/* lista de tareas oredenada */
    insert( S/A, [T/B | L], [T/B | L1], F1, F2) :-
        insert(S/A, L, L1, F1, F2).

    insert(S/A, [], [S/A], _, A).
    insertidle( A, [T/B | L], [idle/B | L1]) :- A < B, !.
    insertidle(A, [T/B | L], [T/B | L1]) :-
        insertidle(A, L, L1).
    del( A, [A | L], L).       /* borra un elemento de la lista */
    del( A, [B | L], [B | L1]) :-
        del( A, L, L1).
    goal([] * _ * _).       /* el estado meta no es una tarea en espera */

    /*
La estimación heurística de la asignación parcial de tareas se basa en una es-
timación optimista del tiempo de término */

    h( Task * Proccesor * Fin, H) :-
        totaltime(Tasks, Tottime),      /* duración total de las tareas en es-
pera */
        sumnum( Processor, Ftime, N),

    /* El tiempo F es la suma de los tiempos que tardan los N procesadores en
realizar las tareas asignadas */
        Finall is (Tottime + Ftime)/N,
        (Finall > Fin, !, H is Finall - Fin
        ;
        H = 0 ).

    totaltime([], 0).
    totaltime([_/D | Tasks], T) :-
        totaltime(Tasks, T1),
        T is T1 + D.

    sumnum([], 0, 0).
    sumnum([_/T | Procs], FT, N) :-
        sumnum(Procs, FT1, N1),
        N is N + 1,
        Ft is FT1 + T.
```

```
/* la gráfica de precedencias de la instancia del problema 13 de la sección
5.3 */
prec(t1, t4).      prec(t1, t5).
prec(t2, t4).      prec(t2, t5).
prec(t3, t5).      prec(t3, t6).
prec(t3, t7).

/* un estado inicial */
start([t1/4, t2/2, t3/2, t4/20, t5/20, t6/11, t7/11] *
    [idle/0, idle/0, idle/0] * 0)

/* ejemplo de una pregunata */
start(Problem), a-star(Problem, Sol).
```

APÉNDICE B

9. REFERENCIAS

[Aho, Hopcroft, Ullman (1974)]
A. Aho, J. Hopcroft, J. Ullman, *Design and Analysis of Computer Algorithms,* **Addison Wesley 1974.**

[Aigner(1988)]
M. Aigner, *Combinatorial Search,* **John Wiley & Sons, and B. G. Teubner, Stuttgar 1988.**

[Aleksander, Ferreny, Ghallab (1986)]
I. Aleksander, H. Ferreny, M. Ghallab, *Decision and Intelligence,* **London: Kogan Page 1986.**

[Amarel(1968)]
S. Amarel, *On Rpresentation of Problems of Reasoning about actions,* **Machine Intelligence, 1968, nr 3, pp. 131-171.**

[Aarts, Laarhoven(1988)]
E.H. Aarts,P.J.Laarhoven, *Simulated Annealing: Theory and Aplications* **Kluwer Acadmic Publishers.**

[Basse(1988)]
S. Basse, *Computer Algorithms: Introduction to design and analysis,* **2nd., Addison Wesley 1988.**

[Bellman, Cooke, Lockett (1970)]
R. Bellman, K. Cooke, Lokett, *Algorithms, Graphs and Computers,* **Academic Press 1970.**

[Bolc, Cytowski (1992)]
L. Bolc, J. Cytowski, *Search Methods for Artificial Intelligence,* **Academic Press 1992.**

[Brookshear(1989)]
J. G. Brookshear, *Theory of Computation, Formal Languages, Automata and Complexity,* **Benjamin Cummings Publishing 1989.**

[Chang, Lee (1973)]
C.Chang, R. Lee, *Symbolic Logic and Mechanical Theorem Proving,* **Academic Press 1973.**

[Even(1979)]
S. Even, *Graph Algorithms,* **Computers Science Press, Potomac. MD. 1979.**

[Cohen(1987)]
D. Cohen, *Computability and Logic,* **Ellis Horwood Limited and John Wiley 1987.**

[Cook(1971)]
S. A. Cook, *The complexity of theorem-proving procedures,* **Proc. Third ACM Symposioum on Theory of Computing, 1971, pp. 151-158.**

[Cormen, Leirserson, Rivest (1990)]
T. Cormen, C. Leirserson, R. Rivest, *Introduction to Algorithms,* **McGraw-Hill.**

[Fang, Puthenpura (1993)]
S.Fang, S.Puthenpura, *Linear Optimization and extentions: Theory and Algorithms,* **Prentice Hall 1993.**

[Feigenbaum, Feldman (1963)]
E. Feigenbaum, J. Feldman, *Computer and Thought,* **New York, McGraw-Hill 1963.**

[Foulds(1992)]
L. R. Foulds, *Graph Theory Applications,* **Springer-Verlag, New York 1992.**

[Gallier(1986)]
J. Gallier, *Logic for Computer Science: Foundations of Automatic theorem proving,* **Harper & Row Publishers, 1986.**

[Garey, Johnson (1979)]
M. Garey, D. Johnson, *Computer and Intractability: A Guide to the theory of NP-completeness,* **Freeman 1979.**

[Gould(1988)]
R. Gould, *Graph Theory,* **Benjamin Cumming Publishing 1988.**

[Guida, Somalvico (1979)]
G. Guida, M. Somalvico, *A Method for computing Heuristic in Problem Solving,* **Information Science 1979, nr 19, pp. 251-259.**

[Harary(1971)]
F. Harary, *Graph Theory,* **Addison Wesley, Reading Mass. 1971.**

[Hermes(1969)]
H. Hermes, *Enumeratibility, Decibility, Computability,* **Springer Berlin 1969.**

[Hopcroft, Ullman (1979)]
J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages and Computation,* **Addison- Weslwy 1979.**

[Horst(1986)]
P. Horst, *A general class of branch and bound method in global optimization wiht some new approaches for concave minimization,* **J. Optimiz. Theory and Applic. 1986, 51, No., 2, pp. 271-291.**

[Hu(1982)]
T.C. Hu, *Combinatorial Algorithms,* **Addison Wesley.**

[Jackson(1986)]
P. Jackson,*Introduction to Expert Systems,* **Addison Wesley.**

[Latombe(1991)]
J. C. Latombe,Motion Planning, Kap, Stanford Press.

[Mahanti,Ghosh (1991)]
A. Mahanti, S. Ghosh, *Controlled Best-First Search for near Optimal Solutions,* **IEEE 1991, pp.109-118.**

[Mahmoud(1992)]
H. Mahmoud, *Evolution of Random Search Trees,* **John Wiley 1992.**

[Martelli(1977)]
A. Martelli, *On the search complexity of admisible search algorithms,* **Artificial Intelligence 1977, vol. 8, pp. 1-13.**

[McHugh(1990)]
J. McHugh, *Algorithmic Graph Teory,* **Prentice-Hall 1990.**

[McNaughton(1982)]
R. McNaughton, *Elementary Computability: Formal languages and automata,* **Prentice-Hall 1982.**

[Nemhauser, Wosley (1988)]
G. Nemhauser, L. Wosley, *Integer and Combinatorial Optimization,* **JohnWiley 1988.**

[Newell(1969)]
A. Newell, *Heuristic Programing Ill-structured problems in: Progress in Operation Research III,* **Aronofsky, J.S. (Ed), New York, John Wiley 1969.**

[Nilsson(1971)]
N. Nilsson, *Problem Solving: methods for artificial intelligence,* **McGraw-Hill 1971.**

[Nilsson(1980)]
N. Nilsson, *Principles of Artificial Intelligence,* **Tioga 1971.**

[Papadimitriou(1994)]
C. Papadimitriou, *Computational Complexity,* **Addison- Wesley, Reading Ma. 1994.**

[Papadimitriou, Steiglitz (1982)]
C. Papadimitriou, K. Steiglitz, *Combinatorial Optimization: algorithms and complexity,* **Prentice-Hall 1982.**

[Passino, Antsaklis (1994)]
K. Passino, P. Antsaklis, *A metric space approach to the specification of the heuristic function for theA* algorithm,* **Transaction on systems, man and Cybernetics, January1994, vol. 24, No. 1.**

[Pearl(1983)]
J. Pearl, *Heuristics: Intelligence search strategies for computer problem solving,* **Addison Wesley 1983.**

[Polya(1964)]
G. Polya, *How to solve it?,* **New York Princeton 1964.**

[Rawlins(1992)]
G. Rawlins, *Compared to What?: An introduction to the analysis of algorithms,* **Freeman 1992.**

[Reeves(1992)]
C. Reeves, *Modern Heuristic Techniques for Combinatorial Problems,* **John Wiley 1992.**

[Reingold, Nievergelt, Deo (1977)]
E.Reingold, J.Nievergelt, N.Deo, *Combinatorial algorithms: Theory and practice,* **Prentice Hall 1977.**

[Rich(1991)]
A. Rich, *Artificial intelligence,* **2nd. ed., McGraw-Hill 1991.**

[Shirai, Tsujii (1982)]
Y. Shirai, J. Tsujii, *Artificial Intelligence: Techniques and aplications,* **John Wiley 1982.**

[Solama(1985)]
A. Solama, *Computation and automata,* **Cambrige University Press 1985.**

[Winston(1977)]
P. Winston, *Artificial Intelligence,* **Addison-Wesley 1977.**