Capítulo 1

Conceptos Básicos

1.1. Qué es Inteligencia Artificial

1.2. Complejidad de problemas

1.2.1. Presentación y definiciones

Problema del Vendedor Viajero

Para ilustrar este capítulo se va a trabajar en base al problema de teoría de grafos ampliamente conocido: El Problema del Vendedor Viajero[1].

Dado un grafo completo con "n" nodos, donde cada arco tiene asociado un largo, se busca un ciclo hamiltoneano de largo mínimo del grafo. Un algoritmo trivial sería: Enumerar todos los ciclos hamiltoneanos del grafo y calcular su largo, de forma tal de poder determinar el de menor largo. Lamentablemente este algoritmo es imposible de llevar a cabo en la práctica, aún considerando un número reducido de nodos: Un grafo completo con n nodos posee $\frac{(n-1)!}{2}$ ciclos hamiltoneanos. Por ejemplo para $n = 20$ se requeriría alrededor de 19 siglos de cálculo en un computador capaz de generar $1{,}000{,}000$ de ciclos hamiltoneanos por segundo. Se puede acelerar la resolución usando un algoritmo tipo branch and bound, sin embargo todos los algoritmos exactos conocidos actualmente requieren un tiempo de cálculo demasiado importante como para que sea razonable el aplicarlos, de un punto de vista práctico, cuando el tamaño del grafo se vuelve un poco elevado. Vamos a ver que este problema no es el único para el cual no se conoce un "buen" algoritmo de resolución exacta y que parece casi imposible el encontrar uno algún día (no significa que a priori este no exista). Veamos algunas definiciones previas.

[1] En inglés Travelling Salesman Problem

1.2.2. Tamaño de una Instancia

Para definir el tamaño de una instancia se requiere tomar en cuenta todos los factores. Por ejemplo, en el caso de una instancia para el problema del camino más corto se debe tomar en cuenta no solo el número de nodos del grafo sino también de las distancias entre los pares de nodos. La descripción de una instancia de un problema puede ser visto como una cadena de caracteres que pertenecen a un alfabeto fijo. El largo de esta cadena de caracteres definirá el tamaño de la instancia. El alfabeto binario permite una codificación razonable. De esta forma representar un número entero k positivo con esta codificación requerirá $round(log_2(k+1))$ bits; representar un grafo de n nodos por su matriz de adyacencia necesitará tantos bits como pares de nodos, es decir n^2. En este sistema binario, el tamaño de la instancia es el número de bits necesarios para representar todos los datos que permiten definir la instancia, tratándose ya sea de números, conjuntos, grafos u otra estructura. Se adopta esta codificación para los estudios de complejidad.

1.2.3. Máquina de Turing y complejidad de un algoritmo

Para definir la noción de complejidad de un algoritmo se define primero lo que es una *Máquina de Turing determinista de una banda*, esta máquina está compuesta de tres partes:

- Un conjunto de estados de la máquina, constituyendo una "unidad central" que controla el conjunto de ruteo del programa, con la ayuda de una "tabla" que contiene la descripción de las operaciones a realizar.
- de una banda infinita constituida de casillas numeradas $\ldots, -3, -2, -1, 0, 1, 2, \ldots$ sobre la cual están inscritos los datos y donde serán inscritos los cálculos intermedios y los resultados finales.
- De una cabeza de lectura-escritura susceptible de leer y de modificar las informaciones que figuran sobre la banda, de acuerdo a las instrucciones entregadas por la "unidad central".

El comportamiento de la máquina de Turing depende de la tabla almacenada en la unidad central y de los símbolos leídos en las casillas de la banda. En cada paso, la cabeza de lectura-escritura lee el símbolo contenido en la casilla de la banda sobre la cuál está apuntando. En función de este símbolo y del estado actual de la máquina, la tabla indica a la cabeza lo que debe escribir en la casilla sobre la que está apuntando, determina el desplazamiento de la cabeza (una casilla en un sentido o en el otro, a menos que reste inmóvil) y define el estado siguiente de la máquina.

El siguiente ejemplo ilustra el funcionamiento de una máquina de Turing y la modelización de un programa por este tipo de máquina, esto nos permite dar una definición precisa de lo que son los algoritmos polinomiales. La máquina está constituída:

1. De un conjunto finito de estados Q compuesto de un estado inicial q_0, de dos estados finales q_s y q_e (s de éxito y e de fracaso; si se aplica esta máquina de Turing a un problema de reconocimiento, es decir a un problema donde uno

hace una pregunta y cuya respuesta es “si” o “no”, el éxito se podrá interpretar como una respuesta “si” y el fracaso como un “no”). Se incorporan además los estados q_1 y q_2.

2. De un conjunto finito de valores que se pueden leer o escribir en las casillas de la banda, que contienen una palabra de $\{0,1\}^*$(si A representa un alfabeto, es decir un conjunto de símbolos, A^* indica el conjunto de palabras que se pueden escribir a partir de A, es decir toda cadena de símbolos que pertenece a A) y un símbolo especial que se llamará $blanco\ b$.

3. De una función de transición que en un estado distinto de q_s y q_e tiene asociada una tripleta cuyo primer elemento indica el nuevo estado de la máquina, el segundo caracter reemplazará al caracter leído, el tercero indica el sentido de desplazamiento de la cabeza de la lectura (desplazamiento de a lo mas una casilla: se denotará por $+1$ un desplazamiento hacia la derecha, -1 un desplazamiento hacia la izquierda, y 0 la ausencia de desplazamiento); la máquina se detiene en alguno de los estados q_s o q_e.

La tabla indica las transiciones de la máquina (o programa) considerada. El índice de la línea corresponde al estado de la máquina, el índice de la columna al caracter leído sobre la banda.

	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_e, b, 0)$	
q_2	$(q_s, b, -1)$	$(q_e, b, 0)$	$(q_e, b, 0)$

El efecto de ese programa es el siguiente: el dato es una cadena X de símbolos pertenecientes a $\{0,1\}$. La cadena X está localizada sobre la banda, un símbolo por casilla, el primero en la posición 1, el último en la posición $|X|$. Las otras casillas de la banda contienen inicialmente b. La máquina está inicialmente en el estado q_0 , y la cabeza de lectura-escritura está ubicada sobre la casilla número 1. El cálculo se realiza etapa por etapa. Es fácil ver que el estado q_0 consiste a desplazarse hacia la derecha en la búsqueda de un blanco, o sea al final de la palabra, sin cambiar nada. Cuando se encontró el primer blanco, se cambia de estado y se pasa a q_1 devolviéndose a una casilla anterior: Se apunta, a partir de ahora, sobre el último bit de la palabra. Si este es un “1”, se detiene en el estado q_e: fracaso; si se lee un “0” se pasa al estado q_2 y se devuelve una casilla más (debe notarse que no es posible leer un blanco). Si el nuevo bit es un “0” se detiene en el estado q_s: la respuesta será “si”; en los otros dos casos se alcanza el estado final q_e: la respuesta es “no”. Finalmente, se ve que esta máquina modela un programa que prueba que si una palabra de $\{0,1\}^*$ se termina por un 00: esta es una condición necesaria y suficiente para que sea reconocido por la máquina, o para que la máquina termine el cálculo en el estado q_s.

Un problema de reconocimiento se puede traducir en términos de la máquina de Turing determinista (una banda). Por ejemplo, la máquina de Turing anterior está asociada al siguiente problema: dado un entero $N > 0$, existe un entero m tal que $N = 4m$?. La codificación asociada a una instancia del problema es simple, ya que basta con dar

una representación binaria a N. La máquina de Turing definida se termina para todo dato (siguiendo los elementos iniciales en la banda): el cálculo se termina en el estado "éxito" si los dos bits a la derecha son 0, ella se termina en "fracaso" sino.

Ahora se puede dar una definición precisa de la complejidad de la máquina M, o del algoritmo modelado por M. Si, para todo dato, se alcanza un estado final (q_e ó q_s) en un número finito de etapas, se puede definir un "tiempo de cálculo" o complejidad como el número máximo de pasos efectuados por M para poder tratar cualquier dato, de tamaño fijo del problema. Este número de pasos o este tiempo de cálculo $T_M(n)$ es una función del tamaño $n = |X|$ del dato X. Si existe un polinomio p tal que $T_M(n) \leq p(n)$, se habla de una máquina de Turing polinomial y se dice que el algoritmo modelado por M es polinomial (lo que no significa que su complejidad se pueda expresar por un polinomio con respecto al tamaño de los datos, sino solamente que se puede acotar por tal polinomio; asi un algoritmo cuya complejidad vale $log_2 n$, donde n es el tamaño de los datos, es polinomial). Un algoritmo cuya complejidad no es acotable por un polinomio con respecto al tamaño de los datos se dice *exponencial* , aunque su complejidad no se exprese a través de una exponencial en el sentido matemático del término (por ejemplo, un algoritmo de complejidad $n!$, donde n es el tamaño del dato, es exponencial: la función factorial no es acotable por un polinomio en n, sin ser ella misma una función exponencial). A veces se da el nombre de "buenos algoritmos" o de "algoritmos eficaces" a los algoritmos polinomiales; sin embargo, en la práctica sucede que algoritmos exponenciales muestran una mejor performance.

En la práctica, esta definición de complejidad de un algoritmo es reemplazada por una noción más fácil de manipular. Se sustituye el número de pasos efectuados por una máquina de Turing por el número de operaciones consideradas elementales efectuadas por un computador secuencial; se entiende por elemental una operación modelada en un número polinomial (con respecto al tamaño de los datos) de pasos de una máquina de Turing: Este es el caso por ejemplo de las cuatro operaciones aritméticas, de las diferentes operaciones de comparación y de asignación.

1.2.4. Problemas de Reconocimiento

Los problemas de reconocimiento se conocen también como problemas de decisión, es decir de enunciados para los cuales la respuesta es "si" o "no" y no a problemas de optimización. Existe una unión importante entre un problema de reconocimiento y problema de optimización, esto se ilustrará con el problema del vendedor viajero. El problema del vendedor viajero (TSP) se puede escribir de la siguiente manera (para simplificar el desarrollo se supondrá que los valores del grafo son enteros y positivos):

Nombre: TSP

Datos: Dado un grafo G de orden n, completo y valorizado con valores positivos

Objetivo: Determinar el largo mínimo de un ciclo hamiltoniano en el grafo G

El problema de reconocimiento (RTSP) asociado al TSP se puede enunciar como:

Nombre: RTSP

Datos: Dado un entero k, dado un grafo G de orden n, completo y valorizado con valores positivos

Pregunta: Existe en el grafo un ciclo hamiltoniano de largo inferior o igual a k?

Es obvio que si se resuelve TSP para el grafo G, se resuelve inmediatamente RTSP. En consecuencia, el tamaño de las instancias de RTSP y de TSP difieren entre ellas sólo por el término $round(log_2(k+1))$, si existe un algoritmo polinomial (con respecto al tamaño de las instancias de TSP) para resolver TSP, entonces existe un algoritmo polinomial (con respecto al tamaño de las instancias de RTSP) para resolver RTSP. Suponga que se resuelve sucesivamente un número finito de instancias de RTSP, dándole valores enteros a k, disminuyendo de 1 a cada etapa. Partiendo de un valor inicial de k, para el cual se está seguro que la respuesta es "si", el último valor de k para el cual la respuesta es "si" entrega el valor mínimo del ciclo hamiltoniano. Se puede realizar un procedimiento mejor como utilizar la dicotomía para encontrar el mínimo: Se comienza al igual que antes, dando un valor a k para el cual la respuesta es "si", por ejemplo $n.val_{max}$ donde val_{max} denota el valor más grande del grafo. Se resuelve RTSP asignando $k = int(\frac{val_{max}}{2})$: si la respuesta es "si" se sigue con $k = int(\frac{val_{max}}{4})$, sino con $k = int(\frac{3val_{max}}{2})$ y así sucesivamente. Asi en las partes enteras cercanas el intervalo de búsqueda, disminuyen a la mitad y bastan alrededor de $log_2(nval_{max})$ resoluciones de instancias de RTSP para resolver la instancia asociada al TSP. Luego, todo algoritmo A_R de complejidad $c(A_R)$ que permite resolver RTSP define un algoritmo A_0 de complejidad del orden de $log_2(nval_{max})c(A_R)$ que permite resolver TSP. O si val denota la función de evaluación de G, el tamaño de una instancia de TSP es del orden de $T_0 = \sum_{\{i,j\}\in G} log_2(val(i,j)+1)$, ya que se deben codificar los enteros $val(i,j)$ para todos los arcos (i,j) del grafo (se debe notar que esta cantidad es superior o igual a $\frac{n(n-1)}{2}$ superior a $log_2 n$) y el tamaño de la instancia de RTSP es del orden de $T_R = log_2(k+1) + \sum_{\{i,j\}\in G} log_2(val(i,j)+1)$. Dándose cuenta que se está interesado sólo en las instancias de RTSP para las cuales $k < nval_{max}$ (las otras instancias tienen la respuesta trivial "si"), es fácil mostrar el siguiente resultado: si existe un polinomio p en T_R que acota la complejidad de A_R, luego existe un polinomio q en T_0 que acota la complejidad de A_0. Dicho de otra forma, si existe un algoritmo polinomial (con respecto al tamaño de las instancias de RTSP) para resolver RTSP, luego existe un algoritmo polinomial (con respecto al tamaño de las instancias de TSP) para resolver TSP.

Estos resultados muestran la unión entre RTSP y TSP: uno permite resolución en tiempo polinomial si y sólo si el otro también lo permite. Esta propiedad, general, nos permite interesarnos en los problemas de reconocimiento para obtener información de la dificultad de los problemas de optimización a los cuales éstos están asociados.

1.2.5. Clases P y NP; problemas NP-completos

Las nociones presentadas en esta sección se refieren a problemas de reconocimiento, sin embargo, como se mencionó anteriormente, los resultados obtenidos nos darán informaciones relacionadas con los problemas de optimización.

La clase P

Definición 1.2.1 *Un problema se dice polinomial si existe un algoritmo de complejidad polinomial que permite responder la pregunta del problema cualquiera sea el dato de éste. La clase P es el conjunto de todos los problemas de reconocimiento polinomiales.*

Se conocen varios problemas que son polinomiales. Por ejemplo los problemas de programación lineal, aunque no sea el algoritmo simplex que permite responder la pregunta. Sin embargo, no conocer un algoritmo polinomial que resuelva un problema dado no significa que no exista. Para tener en cuenta esta dificultad, se define una clase de problemas mas amplia, la clase NP.

La clase NP

Definición 1.2.2 *Un problema de reconocimiento está en la clase NP si, para toda instancia de ese problema, se puede verificar, en un tiempo polinomial con respecto al tamaño de la instancia, que una solución propuesta o adivinada permite afirmar que la respuesta es "si" para esta instancia.*

Se debe notar que no nos interesa la justificación de una respuesta negativa: se busca sólo verificar una respuesta "si". Veamos el problema de vendedor viajero

Proposición 1.2.1 *RTSP está en la clase NP*

Prueba: Basta demostrar cómo verificar, en un tiempo polinomial, que una posible solución permite establecer la respuesta "si" para la instancia considerada. Pensemos que alguien nos dice que la respuesta de esta instancia es "si" y, para convencernos de este resultado presenta una manera de visitar las ciudades diciendo que es un ciclo hamiltoneano de largo inferior o igual a una cota k. Habría entonces que verificar dos cosas:

1. La estructura propuesta es un ciclo hamiltoneano
2. El largo de ese ciclo es inferior o igual a k

Para ello, verificamos primero que la secuencia S de nodos propuestos como solución contiene el número n de nodos: luego, se crea una tabla T con n casillas enumeradas por nodo e inicializada en 0; luego se recorren nuevamente los nodos de S y se incrementa de acuerdo a los valores contenidos en T: Al final, los valores de T entregan el número de ocurrencias de los nodos en S. Es fácil deducir asi si S es un ciclo hamiltoneano. Se puede también durante el recorrido calcular el largo de S; comparándolo con k, se verifica de esta manera la parte 2. Todas estas verificaciones necesitan un número de operaciones $O(n)$ lo que es acotable por un polinomio $log_2(k+1) + \sum_{\{i,j\} \in G} log_2(val(i,j)+1)$, el orden del tamaño de la instancia RTSP. Se pudo asi verificar en un tiempo polinomial que una posible solución entrega correctamente la respuesta "si". En consecuencia RTSP está en NP. Por otro lado, veamos un problema polinomial. Siempre es posible verificar que la respuesta es "si" en un tiempo polinomial, para cualquier instancia: Basta con resolver en un tiempo polinomial,

el problema para la instancia considerada y ver si la respuesta es “si” o “no”. De ahi la siguiente proposición:

Proposición 1.2.2 *Proposición:* $P \subseteq NP$

Se han definido las clases P y NP. No se sabe si estas dos clases coinciden o si la inclusión de la clase P en las clase NP es estricta. De todas maneras estamos interesados en los problemas NP considerados como los más difíciles de esta clase, los problemas NP-completos

Problemas NP-Completos

Transformación polinomial

Una noción fundamental en la teoría de la NP-completitud es la de la *transformación polinomial*: dados dos problemas de decisión D_1 y D_2, se cumplirá que $D_1 \prec D_2$ si se cumplen las siguientes condiciones:

1. Existe una aplicación f que transforme cualquier instancia I de D_1 en una instancia $f(I)$ de D_2, y un algoritmo polinomial, con respecto al tamaño de I para calcular $f(I)$.
2. Hay una equivalencia entre los dos enunciados “D_1 acepta la respuesta “si” para la instancia I” y “D_2 acepta la respuesta “si” para la instancia $f(I)$”.

Si $D_1 \prec D_2$ y si existe un algoritmo polinomial para resolver D_2, luego existe un algoritmo polinomial para resolver D_1. Para obtener la respuesta de la instancia I de D_1, basta con transformar I polinomialmente en la instancia $f(I)$ de D_2 y resolver ésta, siempre polinomialmente: se resuelve I resolviendo $f(I)$. Se puede interpretar intuitivamente la relación $\prec$ como que D_1 no es más dificil que D_2. Esta es una noción transitiva, es decir si $D_1 \prec D_2$ y $D_2 \prec D_3$, luego $D_1 \prec D_3$.

Definición 1.2.3 *Un problema Q se dice NP-completo si pertenece a la clase NP y si, para todo problema Q' de la clase NP, se tiene que $Q' \prec Q$.*

Esta definición tiene varios aspectos importantes: si un problema NP-completo se comprueba polinomial, entonces $P = NP$; si Q y Q' son dos problemas de la clase NP, si Q es NP-completo y si $Q \prec Q'$ entonces Q' es NP-completo. Finalmente, nótese que en el caso $P \neq NP$, los problemas NP-completos y los problemas polinomiales no involucran todos los problemas en NP; si $P \neq NP$ se puede mostrar por el contrario que existe un número infinito de clases de problemas de NP de dificultad intermedia con respecto a esas clases, P siendo la clase menos dificil y los problemas NP-completos la clase mas difícil.

1.2.6. Consecuencias de la NP-completitud de un problema

Para qué sirve la demostración, larga y complicada de que un problema es NP-completo?. Primero que nada esta prueba permite evitar perder el tiempo buscando un algoritmo polinomial para resolver el problema: no encontrar no significa ser incompetente. Además, justifica buscar heurísticas , es decir, algoritmos que para la clase de problemas de optimización entreguen un valor “cercano” al óptimo.

1.2.7. Problemas NP-difíciles

Aqui el interés no es sólo en los problemas de reconocimiento como cuando se estudia las clases P, NP y NP completos. Un problema NP difícil es un problema al menos tan difícil como un problema NP-completo.

Proposición 1.2.3 *Sea O un problema de optimización. Si el problema de decisión asociado a O es NP-completo, entonces O es NP-difícil.*

Por ejemplo para el problema del vendedor viajero TSP es NP-difícil.

1.3. Noción de Espacio de Búsqueda

Capítulo 2

Optimización Combinatoria

2.1. Introducción

2.1.1. Problemas de optimización

Idea: Encontrar una solución “factible” y “óptima” de acuerdo a algún criterio definido.

El criterio puede ser calculado sobre una instanciación completa (hoja de un árbol de búsqueda). Puede también estimarse sobre un nodo correspondiente a una instanciación parcial.

Ejemplos de criterio: Scheduling: minimizar la fecha de término. Problema de asignación de recursos: Minimizar el número de recursos utilizados.

Problema de diseño: Minimizar el costo de solución.

También se puede utilizar esta estrategia para expresar un problema sobre-restringido: Encontrar una solución para las restricciones más difíciles que satisface al menos las restricciones prioritarias.

2.2. Algoritmos de optimización con búsqueda de árbol

Criterio a optimizar: Función de costo (ganancia) a minimizar (maximizar).

Principio: Una solución es una hoja del árbol de búsqueda con el menor costo.

Cada nodo del árbol tiene un valor. La función es monótona decreciente en una rama.

Marco general de aplicación: a diversos problemas de optimización.

- Búsqueda de la ruta más corta
- Vendedor viajero Problema: Encontrar un circuito hamiltoniano (visitar todas las ciudades una sola vez y revenir al punto de inicio) de largo mínimo.

  Problema NP-difícil.

Una relajación a este problema es el problema de asignación. Asignar a cada ciudad otra ciudad que será la siguiente en el tour: Se permiten sub-tours.

Se resuelve el problema de asignación, si hay sub-tours se crean sub-ramas que impiden arcos de ese sub-tour.

Valor del nodo: Costo del problema relajado del nodo.

Dificultades: Pueden producirse infinitas ramas o redundancias (varios nodos representando el mismo estado). Idea: intentar evitar esta situación.

2.3. Problemas de optimización combinatoria

2.3.1. Modelos con restricciones para las variables de *Todo-o-nada*.

Sea el problema lineal

$$\text{máx}\, 18X_1 + 3X_2 + 9X_3$$

s.a.

$$\begin{aligned} 2X_1 + X_2 + \quad 7X_3 &\leq 150 \\ 0 \leq \quad X_1 &\leq 60 \\ 0 \leq \quad X_2 &\leq 30 \\ 0 \leq \quad X_3 &\leq 20 \end{aligned}$$

Nuevo requerimiento: Cada X_i puede ser utilizado ya sea en su cota superior o no ser utilizado.

Sea Y_j = fracción de la cota superior U_j utilizada.
Dominio de $Y_j : \{0, 1\}$:binaria

$$\text{máx}\, 18(60Y_1) + 3(30Y_2) + 9(20Y_3)$$

s.a.

$$\begin{aligned} 2(60Y_1) + (30Y_2) + 7(20Y_3) &\leq 150 \\ Y_1, Y_2, Y_3 &= 0 \text{ ó } 1 \end{aligned}$$

$$\text{máx}\, 1080Y_1 + 90Y_2 + 180Y_3$$

s.a.

$$\begin{aligned} 120Y_1 + 30Y_2 + 140Y_3 &\leq 150 \\ Y_1, Y_2, Y_3 &= 0 \text{ ó } 1 \\ &\ldots \text{ Problema lineal.} \end{aligned}$$

2.3.2. Modelos tipo Problema de la Mochila

Problema de optimización combinatoria "puro". Variables binarias, una sola restricción.

$$\text{máx}\, 8X_1 + 3X_2 + 15X_3 + 7X_4 + 10X_5 + 12X_6$$

s.a.

$$\begin{aligned} 10{,}2X_1 + 6X_2 + 23X_3 + 11{,}1X_4 + 9{,}8X_5 + 31{,}6X_6 &\leq 35 \\ X_1, X_2, \ldots, X_6 &= 0 \text{ ó } 1 \end{aligned}$$

Ejemplo: Dadas las monedas de 1, 5, 10 y 25 centavos, formular un modelo tipo "knapsack" para minimizar el número de monedas necesarias para cambiar **q** centavos.

mín $X_1 + X_5 + X_{10} + X_{25}$ (Total de monedas)

s.a.

$$\begin{aligned} X_1 + 5X_5 + 10X_{10} + 25X_{25} &= \textbf{q} \text{ (cambio correcto)} \\ X_1, X_5, X_{25} &\geq 0 \text{ y enteras.} \end{aligned}$$

2.3.3. Modelos de Presupuesto (o de la Mochila Multidimensional)

Restricciones de presupuesto: Límite de total de recursos disponibles consumidos por los proyectos seleccionados, e inversiones en cada período de tiempo no exceden la cantidad disponible.

Ejemplo de restricciones:

$$\begin{aligned} 6X_1 + 2X_2 + 3X_3 + 1X_7 + 4X_9 + 5X_{12} &\leq 10 \text{ (2000-2004)} \\ 3X_2 + 5X_3 + 5X_5 + 8X_7 + 5X_9 + 8X_{10} + 7X_{12} + 1X_{13} + 4X_{14} &\leq 12 \text{ (2005-2009)} \\ 8X_5 + 1X_6 + 4X_{10} + 2X_{11} + 4X_{13} + 5X_{14} &\leq 14 \text{ (2010-2014)} \\ 8X_6 + 5X_8 + 7X_{11} + 1X_{13} + 3X_{14} &\leq 14 \text{ (2015-2019)} \\ 10X_4 + 4X_6 + 1X_{13} + 3X_{14} &\leq 14 \text{ (2020-2024)} \end{aligned}$$

donde X_i = 1 si la actividad i se realiza, 0 sino.
Agregando restricciones de:

1. Mutuamente excluyentes: La actividad 4 no se puede realizar si se realiza la actividad 5 y vice-versa.
Restricción adicional: $X_4 + X_5 \leq 1$

2. Dependencia: La actividad 11 requiere que la actividad 2 se realice.
Restricción adicional: $X_{11} \leq X_2$

2.3.4. Modelos de Set Packing, Covering, Partitioning

Idea: Identificar los objetos que son solución y que pertenecen a subconjuntos específicos.
Diversos puntos de vista en cuanto a las restricciones.

Restricciones Set covering: Requieren que al menos un miembro de la subcolección J pertenezca a la solución.

Restricción Set packing: Requieren que a lo más un miembro de la subcolección J pertenezca a la solución.

Restricciones Set partitioning: Requieren que exactamente un sólo miembro de la subcolección J pertenezca a la solución.

Ejemplo:

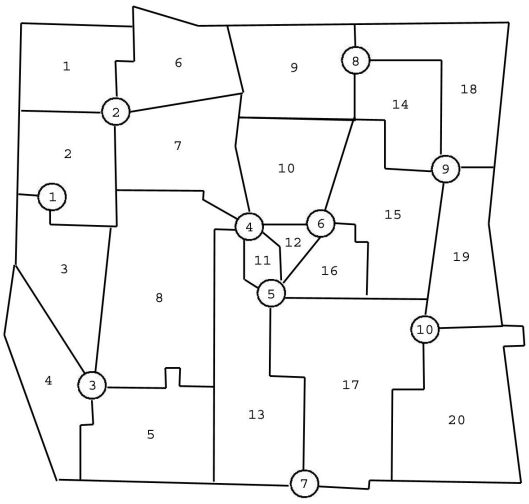


Figura 2.1: Distribución de comunas

Hay 20 comunas y 10 posibles lugares donde instalar equipos de emergencia. Cada lugar puede dar servicio a las comunas adyacentes.

Subcolección 1 =$\{2\}$
Subcolección 2 = $\{1,2\}$
...
Subcolección 12 =$\{4,5,6\}$

Modelo desde el punto de vista Set Covering

mín $\sum X_j$ (j:1..10, número de lugares)
s.a.

$$\begin{aligned} X_2 &\geq 1 \text{ (columna 1)} \\ X_1 + X_2 &\geq 1 \text{ (columna 2)} \\ &\dots \\ X_4 + X_5 + X_6 &\geq 1 \text{ (columna 12)} \\ &\dots \\ X_j &= 0 \text{ ó } 1 \end{aligned}$$

Maximizando covertura

Consideración adicional para el ejemplo:
El presupuesto alcanza para 4 lugares.
Nueva variable Y_i = 1 si la comuna i **no** será servida, 0 en caso contrario.

$$\text{mín} \sum_{i=1}^{20} C_i \cdot Y_i$$

(i: número de comunas, Ci es la importancia de la comuna i)

$$\begin{aligned} X_2 + Y_1 &\geq 1 \\ X_1 + X_2 + Y_2 &\geq 1 \\ &\dots \\ X_4 + X_5 + X_6 + Y_{12} &\geq 1 \\ &\dots \\ \sum_{j=1}^{4} X_j &= 4 \\ X_j &= 0 \text{ ó } 1, \forall j = 1, \dots, 10 \\ Y_i &= 0 \text{ ó } 1, \forall i = 1, \dots, 20 \end{aligned}$$

Ejemplo: Una universidad está adquiriendo un software de programación matemática. Se tiene los siguientes cuatro programas disponibles con los respectivos algoritmos de optimización que incluyen:

Algoritmo	Código,j			
Tipo	1	2	3	4
PL	X	X	X	X
PE	-	X	-	X
PNL	-	-	X	X
Objetivo	3	4	6	14

1. Tomando los coeficientes de la función objetivo como costos, formule un modelo para adquirir un conjunto de programas que provean PL, PE y PNL.

2. Tomando los coeficientes de la función objetivo como costos, formule un modelo para adquirir un conjunto de programas que provean PL, PE y PNL, pero con un presupuesto máximo de 12.

3. Tomando los coeficientes de la función objetivo como costos, formular un modelo para adquirir el conjunto de software de mínimo costo con exactamente uno con PL, uno con PE y uno con PNL.

4. Tomando los coeficientes de la función objetivo como calidad del software, formular un modelo para adquirir el conjunto de software de máxima calidad con a lo más uno con PL, a lo más uno con PE y a lo más uno con PNL.

5. ¿A qué tipo de modelo corresponde cada uno?.

2.3.5. Modelos con Generación de Columnas

La Generación de columnas se utiliza como una estrategia de dos-partes para enfrentar la resolución de problemas combinatorios altamente complejos.

Consiste en la generación de una secuencia de columnas donde cada columna representa una solución factible y luego resolver el problema como un set partitioning (o covering o packing) para seleccionar un conjunto óptimo de esas alternativas.

- Ventaja:flexibilidad.

- Desventaja: Dificultad para enumerar todas las posibilidades.

Ejemplo: AA Crew Scheduling

Posibles secuencias:

j	Secuencia vuelos	Costo	j	Secuencia vuelos	Costo
1	101-203-406-308	2900	9	305-407-109-212	2600
2	101-203-407	2700	10	308-109-212	2600
3	101-204-305-407	2600	11	402-204-305	2600
4	101-204-308	3000	12	402-204-310-211	2600
5	203-406-310	2600	13	406-308-109-211	2600
6	203-407-109	3150	14	406-310-211	2600
7	204-305-407-109	2550	15	407-109-211	2600
8	204-308-109	2500			

Problema: Encontrar la secuencia de vuelos para cada tripulación sobre un período mínimo de 2 días a un máximo de 3 días. La secuencia debe comenzar y terminar en la ciudad donde vive la tripulación. El objetivo es de minimizar costos.

Suponga la siguiente secuencia de viajes de American Airlines.

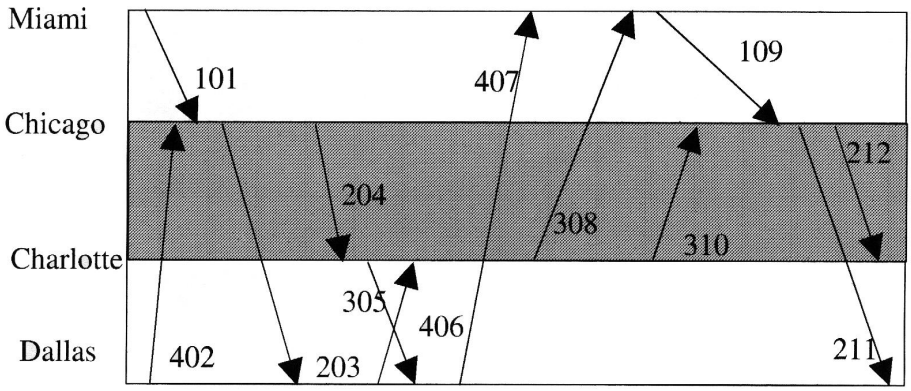


Figura 2.2: Vuelos

$X_j = 1$ si la posibilidad de la columna j se elige, 0 sino.

$$\text{mín}\{2900X_1 + 2700X_2 + 2600X_3 + 3000X_4 + 2600X_5 + 3150X_6$$

$$+2550X_7 + 2500X_8 + 2600X_9 + 2050X_{10} + 2400X_{11} + 3600X_{12}$$

$$+2550X_{13} + 2650X_{14} + 2350X_{15}\}$$

s.a.

$$\begin{aligned} X_1 + X_2 + X_3 + X_4 &= 1 \text{ (vuelo 101)} \\ X_6 + X_7 + X_8 + X_9 + X_{10} + X_{13} + X_{15} &= 1 \text{ (vuelo 109)} \\ X_1 + X_2 + X_5 + X_6 &= 1 \text{ (vuelo 203)} \\ &\dots \\ X_2 + X_3 + X_6 + X_7 + X_9 + X_{15} &= 1 \text{ (vuelo 407)} \\ X_1, \dots X_{15} &= 0 \text{ ó } 1 \end{aligned}$$

Solución óptima: $X_1 = X_9 = X_{12} = 1$, las otras $X_j = 0$, a un costo de 9100.

2.3.6. Modelos de Problemas de Asignación

Encontrar la mejor asignación máquina-trabajo, personal-cliente, etc. Para minimizar costos.

Más simple: Modelo lineal

Sea $X_{ij} = 1$ si i está asignado a j, 0 e.o.c.
Función Objetivo:

$$\sum_i \sum_j C_{ij} \cdot X_{ij}$$

s.a.

$$\begin{aligned} \sum_j X_{ij} &= 1, \forall i \\ \sum_i X_{ij} &= 1, \forall j \\ X_{ij} &= 0 \text{ ó } 1, \forall i, j \end{aligned}$$

Modelos de asignación cuadrática: Quadratic assignment models

Sea C_{ijkl}=costo de asignar i a j y k a l.
Función Objetivo:

$$\sum_i \sum_j \sum_{k \geq i} \sum_{l \neq j} C_{ijkl} \cdot X_{ij} \cdot X_{kl}$$

s.a.

$$\begin{aligned} \sum_j X_{ij} &= 1, \forall i \\ \sum_i X_{ij} &= 1, \forall j \\ X_{ij} &= 0 \text{ ó } 1, \forall i, j \end{aligned}$$

2.4. Problema de Asignación

2.4.1. Modelos de asignación cuadrática: Quadratic assignment models

Ejemplo: Mall layout
Se tienen 4 posibles ubicaciones para departamentos en un shopping mall. Se conocen las distancias (en ft) entre las ubicaciones y las posibles ubicaciones. Se conoce además el número de clientes a la semana que desearían visitar los diferentes pares de departamentos. Por ejemplo se proyecta 5000 clientes a la semana que visitarían la tienda de ropa (1) y computación (2).

Objetivo: Determinar la ubicación de los departamentos minimizando la "molestia" de los clientes.

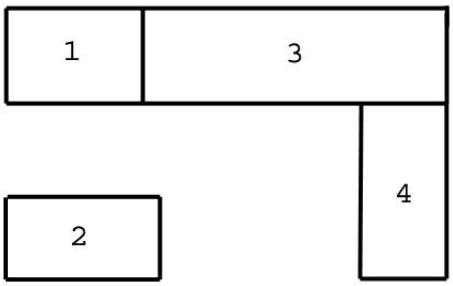


Figura 2.3: Distribución de departamentos

	Distancia [ft]			
Ub(j,l)	1	2	3	4
1		80	150	170
2	80		130	100
3	150	130		120
4	150	100	120	

	Clientes en común			
(Departamento,i)	1	2	3	4
1:Ropa		5	2	7
2:Computación	5		3	8
3:Juegos	2	3		3
4:Libros	7	8	3	

Modelos de asignación general

La asignación de i a j requiere un espacio de tamaño fijo $S_{i,j}$ y la ubicación j tiene una capacidad máxima de B_j.

Sea $X_{ij} = 1$ si i está asignación a j, 0 e.o.c.

Función Objetivo:

$$\sum_i \sum_j C_{ij} \cdot X_{ij}$$

s.a.

$$\begin{aligned} \sum_j X_{ij} &= 1, \forall i \\ \sum_i S_{ij} \cdot X_{ij} &\leq B_j, \forall j \\ X_{ij} &= 0 \text{ ó } 1, \forall i, j \end{aligned}$$

2.4.2. Problema del Vendedor Viajero

Idea: Largo mínimo de la ruta visitando cada punto una sola vez.

TSP simétrico: Si la distancia o costo de pasar desde cualquier punto i a otro j es la misma distancia desde j a i. Sino será un TSP asimétrico.

Modelo para un TCP simétrico:

Sea $X_{i,j} = 1$ si la ruta incluye la secuencia de i ir a j, 0 e.o.c. Para $i < j$.

$$\text{mín} \sum_i \sum_{j>i} D_{i,j} \cdot X_{i,j}$$

s.a.

$$\begin{aligned}
\sum_{j<i} X_{j,i} + \sum_{j>i} X_{i,j} &= 2, \forall i \\
\sum_{i \in S} \sum_{j \notin S, j>i} X_{i,j} + \sum_{i \notin S} \sum_{j \in S, j>i} X_{i,j} &\geq 2, \forall \text{ subconjunto } S, |S| \geq 3 \\
X_{i,j} &= 0 \text{ ó } 1, \forall i; j > i
\end{aligned}$$

Ejemplo:

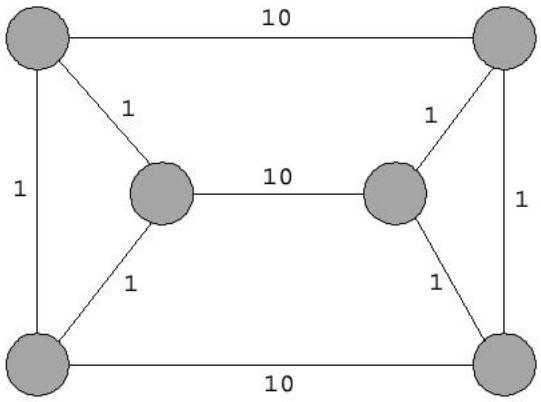
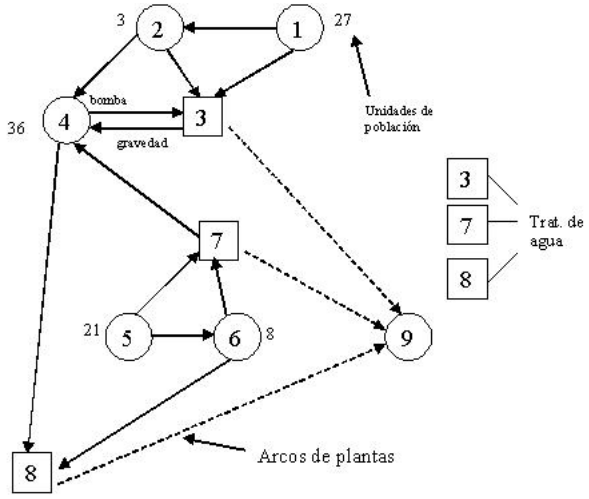


Figura 2.4: Vendedor viajero

$$\text{mín } 10X_{1,2} + 1X_{1,3} + 1X_{1,5} + 1X_{2,4} + 1X_{2,6} + 10X_{3,4} + 1X_{3,5} + 1X_{4,6} + 10X_{5,6}$$

$$\begin{aligned}
X_{1,2} + X_{1,3} + X_{1,5} &= 2 \text{ (nodo 1)} \\
X_{1,2} + X_{2,4} + X_{2,6} &= 2 \text{ (nodo 2)} \\
X_{1,3} + X_{3,4} + X_{3,5} &= 2 \text{ (nodo 3)} \\
X_{2,4} + X_{3,4} + X_{4,6} &= 2 \text{ (nodo 4)} \\
X_{1,5} + X_{3,5} + X_{5,6} &= 2 \text{ (nodo 5)} \\
X_{2,6} + X_{4,6} + X_{5,6} &= 2 \text{ (nodo 6)}
\end{aligned}$$

2.4.3. Modelos del diseño de redes



Arcos	Costo fijo	Costo variable
(1,2)	240	21
(1,3)	350	30
(2,3)	200	22
(2,4)	750	58
(3,4)	610	43
(3,9)	3800	1
(4,3)	1840	49
(4,8)	780	63
(5,6)	620	44
(5,7)	800	51
(6,7)	500	56
(6,8)	630	94
(7,4)	1120	82
(7,9)	3800	1
(8,9)	2500	2

Job-shop Planificación óptima para una colección dada de trabajos c/u requiere una secuencia de procesadores que pueden realizar sólo un trabajo a la vez.

Supongamos 3 trabajos a planificar.

W_1		W_2		W_3	
WS_1	3	WS_7	50	WS_2	5
WS_2	10	WS_1	6	WS_3	9
WS_3	8	WS_2	11	WS_5	2
WS_4	45	WS_3	6	WS_6	1
WS_6	1			WS_4	25

$p_{jk} =$ tiempo en minutos del trabajo j en el procesador k

- ¿Cuándo comenzar?

$X_{jk} \rightarrow$ tiempo de inicio del W_j en el procesador k

- Se desea terminar lo antes posible

$$\text{mín}\{\text{máx}\{X_{12}+1, X_{23}+6, X_{43}+25\}\}$$

- Restricciones de precedencia

$$X_{jk} + p_{jk} \leq X_{jk}$$

- Conflictos

$$Y_{jj'k} = \begin{cases} 1 & \text{Si está antes de } j' \text{ en el procesador } k \\ 0 & \text{e.o.c.} \end{cases}$$

$$\begin{aligned} X_{jk} + p_{jk} &\leq X_{j'k} + M(1 - Y_{jj'k}) \\ X_{j'k} + p_{j'k} &\leq X_{jk} + M \times Y_{jj'k} \end{aligned}$$

2.4.4. Modelo del Problema de ruteo de vehículos VRP

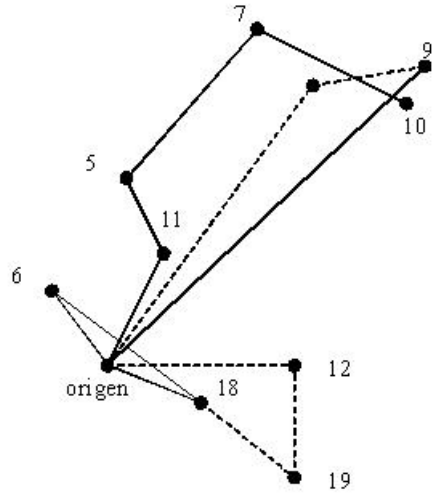


Figura 2.5: Rutas-paradas

Rutas-paradas 20 paradas, 7 rutas.

$$Z_{ij} = \begin{cases} 1 & \text{Si la parada } i \text{ está en la ruta } j \\ 0 & \text{e.o.c.} \end{cases}$$

$$\begin{aligned} \sum_{j=1}^{7} Z_{ij} &= 1 \rightarrow \text{ 20 stops } \forall i = 1, \ldots, 20 \\ \sum_{i=1}^{20} f_i Z_{ij} &\leq 1, \forall j = 1, \ldots, 7 \text{ capacidad camiones } (f_i\text{: \% del camión a entregar}) \\ Z_{ij} &= 0 \text{ ó } 1, \forall j = 1, \ldots, 20; j = 1, \ldots, 7 \end{aligned}$$

Función objetivo:

$$\sum_{j=1}^{7} \Theta_j(z); Z = Z_{ij}$$

Capítulo 3

Problema de Satisfacción de Restricciones CSP

3.1. Introducción

3.2. Definición de un CSP

- Un conjunto de variables: $X = \{X_1, \ldots, X_n\}$.
- Un conjunto de Dominios: $D = \{D_1, \ldots, D_n\}$, donde D_i es el conjunto finito de los valores posibles para X_i.
- Un conjunto de Restricciones: $C = \{C_1, \ldots, C_n\}$, donde C_i está definida sobre un conjunto de variables $\{X_{i1}, \ldots, X_{ik}\}$.
- Un conjunto de relaciones: $R = \{R_1, \ldots, R_n\}$, donde R_i es el conjunto de las combinaciones de valores que satisfacen C_i.

3.3. Representación de un problema

Para un problema binario: las variables y las restricciones forman un grafo.
Para un problema n-ario: las variables y las restricciones forman un hipergrafo.
Representación usando un grafo bipartito:

$$C = \{\{X_1, X_2\}, \{X_1, X_3, X_4\}, \{X_2, X_3, X_4\}\}$$

- Una variable por región.
- Un valor por color.
- Una restricción de desigualdad entre cada región conexa.

3.4. Elección de un Modelo

- Ejemplo sobre el problema de las “n” reinas.
- Problema: Localizar n reinas sobre un tablero de ajedrez $n \times n$ de tal forma que ninguna esté “atrapada”.
- Varias formas de poder modelar el problema
  - Una variable por columna (4^4 posibilidades).
  - Una variable por casilla (2^{16} posibilidades).
  - Una variable por reina (16^4 posibilidades).

3.5. Características de los problemas con restricciones

- Problemas combinatorios complejos (NP-completos).
- El espacio de búsqueda es de tamaño exponencial con respecto al número de variables.
- No se conoce un método de complejidad polinomial para encontrar una solución.

3.6. CSP N-arios a CSP- Binarios

Método de la “Variable Encapsuladora”

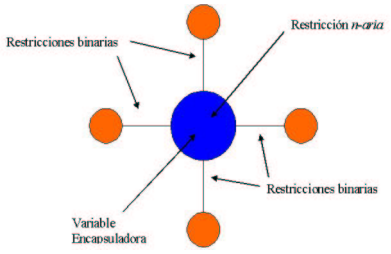


Figura 3.1: Variable encapsuladora

3.7. Ejemplos

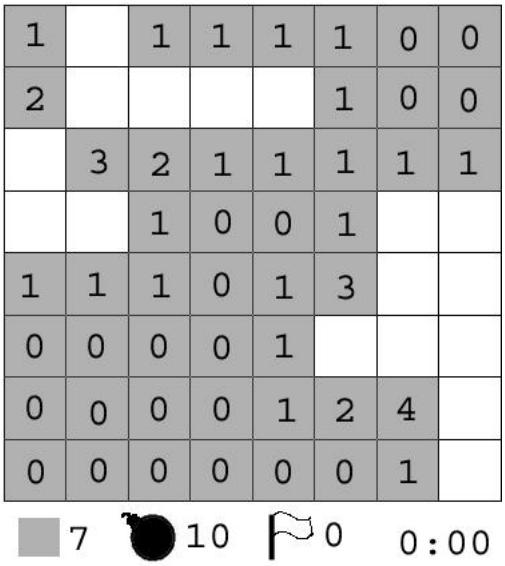


Figura 3.2: Buscaminas

3.7.1. Coloreo de Grafos con 3 colores

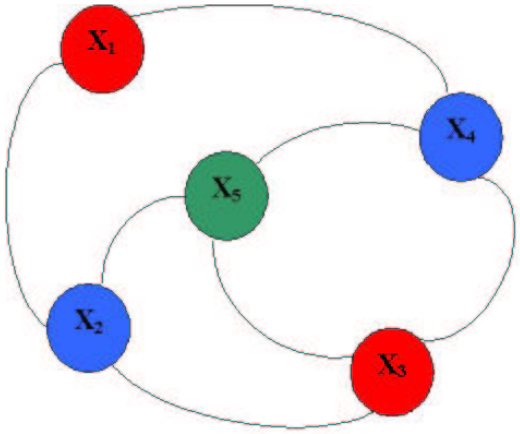


Figura 3.3: Coloreo con 3 colores

En el ejemplo, una solución sería:

$$
\begin{aligned}
X_1 &= Rojo \\
X_2 &= Azul \\
X_3 &= Rojo \\
X_4 &= Azul \\
X_5 &= Verde
\end{aligned}
$$

Capítulo 4

Técnicas completas de Resolución de CSP

4.1. Filtrado y Consistencia

- **Filtrado:**
  1. Transforma un problema P en un P'.
  2. Eliminar los elementos que con *seguridad* no pueden ser parte de la solución.
- **Propiedades:**
  1. Simplificación por reducción del espacio de búsqueda.
  2. El problema reducido es equivalente al problema original.
  3. A veces detecta la ausencia de solución.

Consistencia (coherencia): *Grado de compatibilidad entre los valores de los dominios y las restricciones.*

Varios niveles de consistencia:

- consistencia local (inicial):
  - *Consistencia de arcos*
  - *Consistencia de caminos*
  - *k-consistencia*
- Consistencia Global (resolver)

```
Begin /* Procedure NC(i) */
D_i ⟵ D_i ∩ {x|P_i(x)}
    for i:=1 to n do NC(i)
End
```

Figura 4.1: Consistencia de los nodos

4.1.1. Consistencia en el nodo

Algoritmo de consistencia de nodos

Remark 4.1.1 *Complejidad de $NC(i)$ es $O(an)$, lineal en las variables,*

4.1.2. Consistencia de arcos

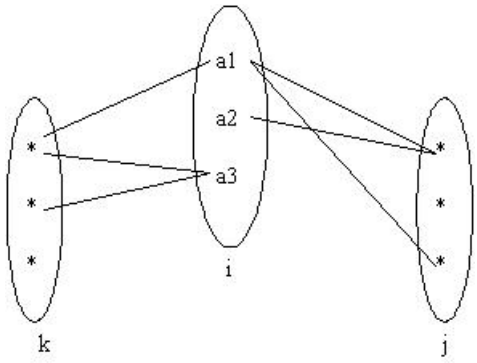


Figura 4.2: Consistencia de Arcos

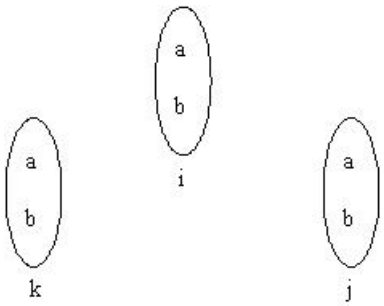


Figura 4.3: Consistencia de Arcos

Definición: Una variable i es *arc-consistente* ssi:
$\forall a \in D_i, \forall j \in V$ conectado a $i, \exists b \in D_j$ tal que $(a, b) \in R_{ij}$
Un problema será arco-consistente si TODAS sus variables son arco-consistentes.
Filtro por consistencia de arco: Eliminar todos los valores que no cumplen con la propiedad.

4.1.3. Consistencia de arcos: AC-3

- Para establecer la consistencia de arcos, se propagan las reducciones de los dominios hasta obtener un punto fijo.
- Un valor es "viable" si posee un valor compatible dentro de los dominios de las variables unidas por una restricción.
- Un valor que "no es viable" será eliminado del dominio de una variable.
- Para realizar la consistencia de arcos:
  1. Sea $Q = \{\text{arc } i \rightarrow j | C_{ij} \text{ rest}\}$
  2. Mientras Q no esté vacío hacer
     a) Elegir y retirar un arco $i \rightarrow j$ dentro de Q
     b) Revisar $i \rightarrow j$
- Para revisar $i \rightarrow j$:
  1. Retirar del dominio de i los valores **a** tales que $\nexists b \in D_j, (a, b) \in R_{ij}$.
  2. Si el dominio de i fue reducido, entonces agregar a Q los arcos $k \rightarrow i$ que unen i a una variable k ($k \neq j$).

1. Procedimiento de consistencia base
2. Algoritmo AC-1
3. Algoritmo AC-2
4. Algoritmo AC-3

```
Begin /* Procedure Revise(i,j) */
BORRAR ⟵ false
    Para cada x ∈ D_i hacer
       Si no hay y ∈ D_j tal que P_ij(x,y) entonces
      begin
          eliminar x del D_i
          BORRAR ⟵ true
      end
    return BORRAR
End
```

Figura 4.4: REVISE

```
Begin /* Procedure AC-1 */
for i:=1 a n do NC(i)
Q ⟵ {(i,j)|(i,j) ∈ arcos(G), i ≠ j}
repeat
        begin
             CAMBIO ⟵ false
             Para cada (i,j) ∈ Q hacer CAMBIO ⟵ (Revise((i,j)) or CAMBIO)
        end
until ¬ CAMBIO
End
```

Figura 4.5: AC-1

4.1.4. Consistencia de arcos: AC-4

Algoritmo basado en la noción de sustento (valores compatibles con un valor dado).
Construcción de apoyos: $S(i,a) = \{(i,j),(j,c),(k,d)\}$
Eliminación de los valores que no tienen apoyo sobre una variable. Propagación inmediata de esta eliminación gracias a la estructura de datos de los apoyos.
Complejidad en espacio $O(ed^2)$
Complejidad óptima en tiempo es $O(ed^2)$

Observación sobre filtrado por consistencia de arcos

- El algoritmo AC-3 no es caro.
- Complejidad en el peor de los casos en tiempo $O(ed^3)$.
- Complejidad en espacio $O(e+nd)$.
- Es simple de implementar.

```
Begin /* Procedure AC-2*/
for i:=1 hasta n do
begin
      NC(i)
      Q ⟵ {(i,j)|(i,j) ∈ arcos(G), i < j}
      Q' ⟵ {(j,i)|(j,i) ∈ arcos(G), j < i}
      While Q ≠ ∅
            begin
            While Q' ≠ ∅
                  begin
                        pop(k,m) desde Q
                        if Revise ((k,m)) then
                          Q' ⟵ Q' ∪ {(p,k)|(p,k) ∈ arcos(G), p ≤ i, p ≠ m}
                  end
                  Q ⟵ Q'
                  Q' ⟵ ∅
            end
end
End
```

Figura 4.6: AC-2

```
Begin /* Procedure AC-3*/
for i:=1 a n do NC(i)
      Q ⟵ {(i,j)|(i,j) ∈ arcos(G), i ≠ j}
      While Q ≠ ∅
            begin
            Seleccionar y borrar un arco(k,m) desde Q
             if Revise ((k,m)) then
               Q ⟵ Q ∪ {(i,k)|(i,k) ∈ arcos(G), i ≠ k, i ≠ m}
             end
End
```

Figura 4.7: AC-3

- Existe AC-5 que es menor en complejidad pero requiere características de biyección, monotonicidad. Esto es utilizado por los softwares actuales.

4.1.5. Consistencia de Caminos

Definición: Un par de variables (i,j) es trayectoria-consistente ssi:
$\forall (a,b) \in D_i \times D_j, \forall k \in V$ conectada a i y j, $\exists c \in D_k, (a,c) \in R_{ik}$ y $(c,b) \in R_{kj}$

Un problema es camino-consistente ssi: Todos los pares de variables son camino-consistentes.

Filtrado por consistencia de caminos: Eliminar todos los pares de valores que no cumplan la propiedad.

Observaciones sobre filtrado por consistencia de caminos

- El algoritmo empieza a ser caro.
- Complejidad en $O(n^3d^5)$ para PC2 (Mackworth 77).
- Complejidad en $O(n^3d^3)$ para PC· (Mohr 86).
- Es más complejo de implementar que AC.
- Su aplicación puede agregar restricciones al grafo, cambia la topología.
- *Conclusión: Poco utilizado.*

4.1.6. K-consistencia

Unifica y generaliza las nociones precedentes de consistencia.

Consistencia de arcos: Consistencia que involucra 2 variables.

Consistencia de caminos: Consistencia que involucra 3 variables.

K-consistencia: Involucra k variables.

Para toda instanciación consiste de k-1 variables, para toda otra variable i, existe un valor para i que forma una instanciación consistente de k variables.

El filtrado por k-consistencia genera restricciones de cardinalidad k-1.

Ejemplo: sea (i,a),(j,b),(k,c),(l,d) una instanciación no-consistente para m. Se crea entonces una restricción de cardinalidad 4 prohibiendo los valores (a,b,c,d).

Complejidad $O(n^k d^k)$.

4.1.7. Ejemplos

Modelización: Coloreo de grafos

- Una variable por región
- Un valor por color
- Una restricción de desigualdad entre cada región conexa

Di={blanco, rosado, rojo, negro}
Restricciones unarias: $X_1 = 1$, $X_2 = 3$, $X_3 \neq$ blanco; $X_3 \neq$ negro.
Restricciones binarias: $X_4 = X_6$, $X_4 = X_5$, $X_5 = X_6$, $X_1 < X_4$, $X_2 < X_4$, $X_3 < X_4$.

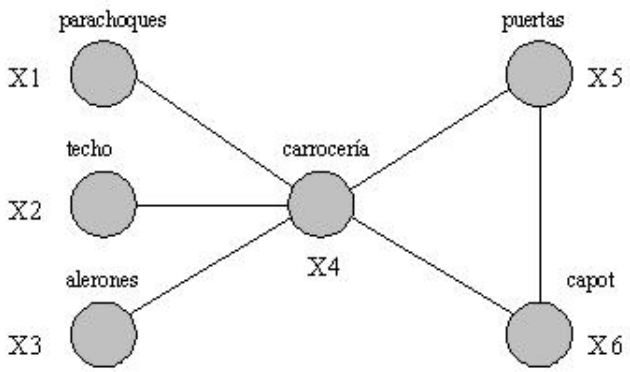


Figura 4.8: CSP

4.1.8. Arco-consistente

1-Ac-consistente: Respeta las restricciones unarias.

$$X_i \in [1, 2, 3, 4]$$

1:blanco
2:rosada
3:roja
4:negra

Restricciones unarias

$X_1 = 1 \Rightarrow X_1 \neq 2, X_1 \neq 3, X_1 \neq 4 \quad D_1 = \{1\}$
$X_2 = 3 \Rightarrow X_2 \neq 1, X_2 \neq 2, X_2 \neq 4 \quad D_2 = \{3\}$
$X_3 \neq$ blanco, $X_3 \neq$ negro $\quad D_3 = \{2, 3\}$

Restricciones binarias

Carrocería=capot=puertas:
$X_4 = X_6, X_4 = X_5, X_5 = X_6$

Más claro que la carrocería:
$X_1 < X_4, X_2 < X_4, X_3 < X_4$

2 Arco-consistente

1. Variable X_1
   C_{14} ? = $X_1 < X_4$
   $D_1 = \{1\} \longrightarrow ?\exists b \in D_4/(1, b) \in R_{14}$; (C_{14} satisfecha)

2. Variable X_2
   C_{24} ? = $X_2 < X_4$

$D_2 = \{3\} \longrightarrow ? \exists b \in D_4/(3,b) \in R_{24}$

3. Variable X_3
   C_{34} ? = $X_3 < X_4$
   $D_3 = \{2,3\}$
   $d_{3,1} = 2 \longrightarrow ? \exists b \in D_4/(2,b) \in R_{34}$
   $d_{3,2} = 3 \longrightarrow ? \exists b \in D_4/(3,b) \in R_{34}$

4. Variable X_4

   a) C_{41} ? = $X_1 < X_4$
   $D_4 = \{2,3,4\}$
   $d_{4,1} = 2 \longrightarrow ? \exists b \in D_1/(1,b) \in R_{41}$
   $\neg\exists \Rightarrow$ eliminar d_{41} $d_{4,2} = 2 \longrightarrow ? \exists b \in D_1/(2,b) \in R_{41}$
   idem resto

   b) C_{42} ? = $X_2 < X_4$
   $D_4 = \{2,3,4\}$
   $d_{4,1} = 2 \longrightarrow ? \exists b \in D_2/(2,b) \in R_{42}$
   $\neg\exists \Rightarrow$ eliminar d_{41} $d_{4,2} = 3 \longrightarrow ? \exists b \in D_2/(3,b) \in R_{42}$
   $\neg\exists \Rightarrow$ eliminar d_{42} $d_{4,3} = 4 \longrightarrow ? \exists b \in D_2/(3,b) \in R_{42}$

   c) C_{43} ? = $X_3 < X_4$
   $D_4 = \{4\} \longrightarrow ? \exists b \in D_3/(4,b) \in R_{34}$

   d) C_{45} ? = $X_4 < X_5$
   $D_4 = \{4\} \longrightarrow ? \exists b \in D_5/(4,b) \in R_{45}$

   e) C_{46} ? = $X_4 = X_6$
   $D_4 = \{4\} \longrightarrow ? \exists b \in D_6/(4,b) \in R_{46}$

5. Variable X_5

   a) $C_{54}? = X_4 = X_5$
   $D_5 = \{1,2,3,4\} \longrightarrow ?$
   $\{1\} \longrightarrow ? \exists b \in D_4/(1,b) \in R_{45}$
   $\neg\exists \Rightarrow$ eliminar d_{51}
   idem d_{52}, d_{53}
   $\Rightarrow D_5 = \{4\}$

   b) C_{56} ? = $X_5 = X_6$
   $D_5 = \{4\} \longrightarrow ? \exists b \in D_6/(4,b) \in R_{56}$

6. Variable X_6

a) $C_{65}? = X_5 = X_6$
$D_6 = \{1, 2, 3, 4\} \longrightarrow ?$
$\{1\} \longrightarrow ?\exists b \in D_5/(1, b) \in R_{45}$
$\neg\exists \Rightarrow$ eliminar d_{61}
idem d_{62}, d_{63}
$\Rightarrow D_6 = \{4\}$

b) C_{64} ? = $X_4 = X_6$
$D_6 = \{4\} \longrightarrow ?\exists b \in D_4/(1, b) \in R_{64}$

$\Rightarrow$ CSP arco consistente:

$$\begin{aligned} X_1 &\in D_1 = \{1\} \\ X_2 &\in D_2 = \{3\} \\ X_3 &\in D_3 = \{2, 3\} \\ X_4 &\in D_4 = \{4\} \\ X_5 &\in D_5 = \{4\} \\ X_6 &\in D_6 = \{4\} \end{aligned}$$

4.2. Técnicas de Búsqueda de soluciones (Diferentes visiones)

- ¿Tiene una solución?
- Encontrar una solución
- Encontrar todas las soluciones
- Encontrar el número de soluciones
- ¿Este valor pertenece a una solución?
- Encontrar todos los valores posibles para una variable
- Encontrar una solución óptima

4.2.1. Técnica Look-ahead

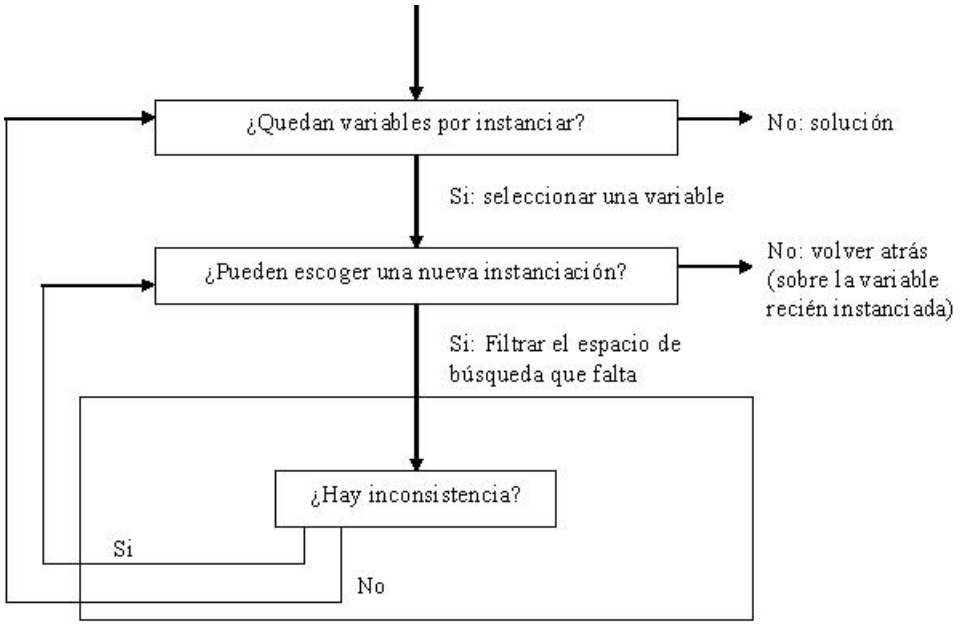


Figura 4.9: Look-ahead

4.2.2. Forward Checking

Se filtran las variables directamente conectadas a la variable instanciada.

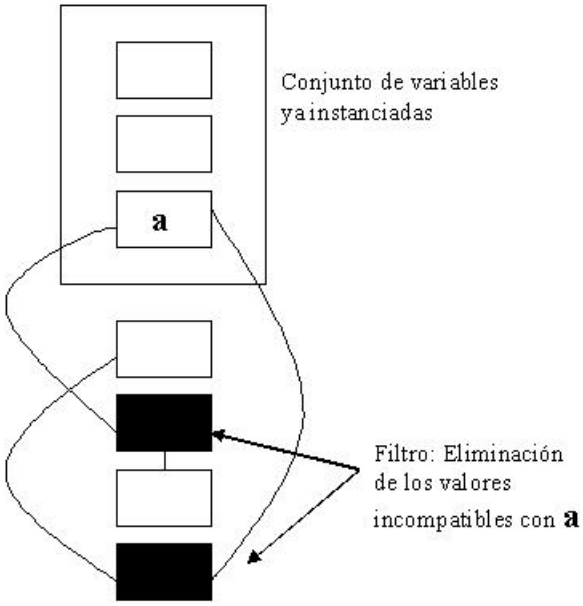


Figura 4.10: Forward checking

4.2.3. Real Full Lookahead

Se filtran por arco-consistencia las variables aún no instanciadas.

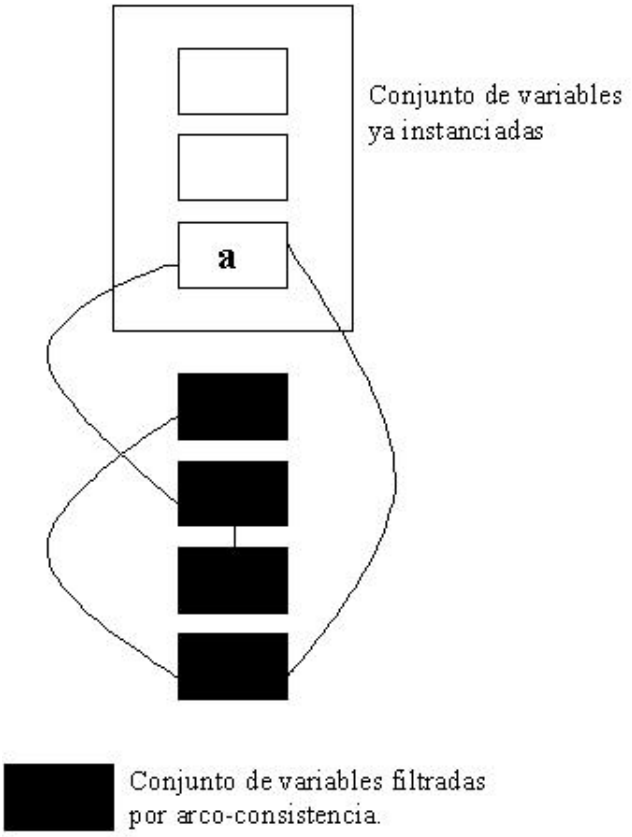


Figura 4.11: Real Full Lookahead

RFL detecta que (1,A),(2,C),(3,E) es un error.

4.2.4. Técnica Look-back

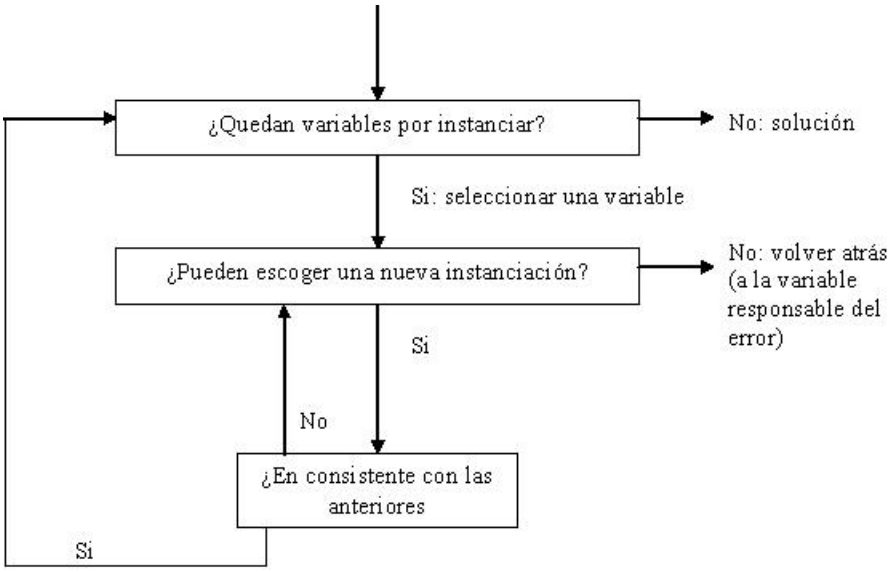


Figura 4.12: Look-back

4.2.5. Técnicas de look-back

GBJ: Retorno guiado por el grafo de restricciones

En caso de error sobre una variable j, volver a la variable i conectada a j por una restricción y la más recientemente instanciada. Método interesante para los grafos de restricciones sparse.

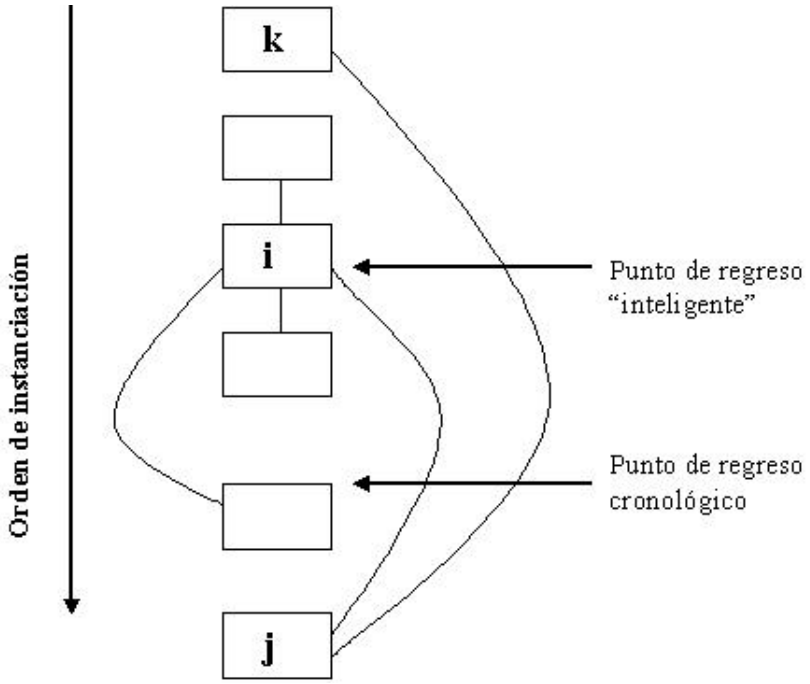


Figura 4.13: GBJ

Ejemplo: 3 colores modificado

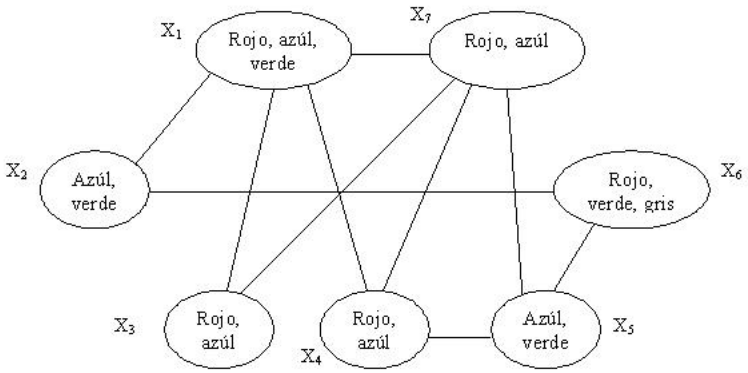


Figura 4.14: 3-color-modificado

1. Construya el árbol de búsqueda que realiza BT.
2. ¿Cambia en algo la búsqueda si decide comenzar con una variable con dominio

más reducido?¿Por qué?

3. Construya la red arc-consistente equivalente. ¿Cuál es el nuevo árbol de búsqueda?

4. ¿Qué hace FC?¿Cuál es el árbol de búsqueda?

5. ¿Qué hace RFL? ¿Cuál es el árbol de búsqueda?

6. ¿Qué hace GBJ?

7. ¿Qué sucede con los cambios de orden en la instanciación?

Backtraking

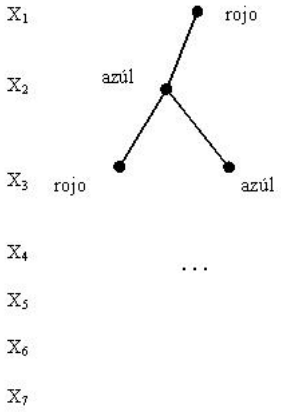


Figura 4.15: Backtraking

CBJ: Retorno guiado por los conflictos

Para cada variable guardar el conjunto de conflictos Conf(I). Para cada valor "erróneo-egistrar en Conf(I) la variable más prematuramente instanciada y en conflicto con el intento actual de instanciación.

Cuando no quedan valores a intentar, el conjunto entrega las causas del problema y el punto de regreso será la variable más reciente g en el conjunto de conflictos.

Se actualiza el conjunto de conflictos.

Ejemplo: Reinas

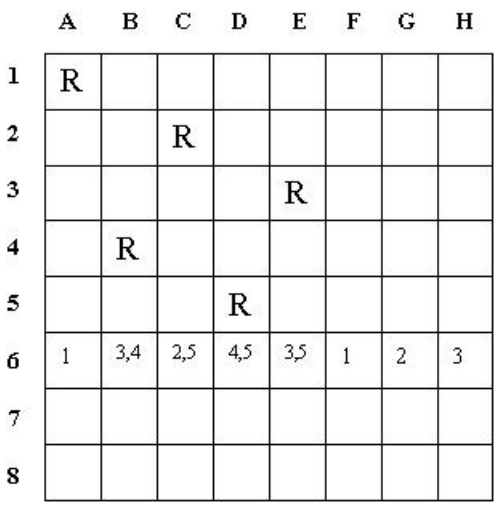


Figura 4.16: 8 reinas

4.2.6. Memorización de Restricciones: Aprendizaje

En caso de error, la instanciación actual constituye un conjunto de valores incompatibles (una restricción) pero es posible que sub-instanciaciones sean también incompatibles.

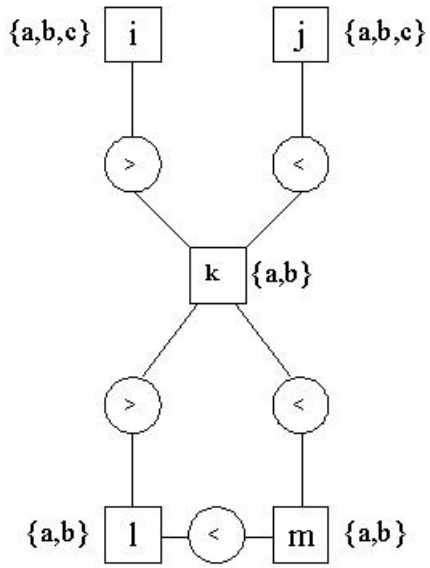
Buscar la sub-instanciación incompatible mínima cuesta caro.

Es también útil limitar el tamaño de las restricciones aprendidas para evitar un costo de memoria importante.

Idea: Buscar una consistencia parcial durante la búsqueda solamente cuando sea necesaria.

Puede modificar el grafo de restricciones.

Ejemplo



Inst:
$< i, b >< j, b >< k, a >< l, b >$
m?
Conflicto:
$S1 = \{< i, b >, < j, b >< k, a >, < l, b >\}$
Conflicto reducido:
$S2 = \{< k, a >, > l, b >\}$
Conflicto mínimo:
$S3 = \{< l, b >\}$
Instanciación: (1,A), (2,C), (3, E)

1. ¿Qué hace FC?

2. ¿Qué hace RFL?

4.2.7. Guiar la Búsqueda

Heurísticas de elección:

Vertical: elección de las variables.

Horizontal: elección de los valores.

Elección de los valores:

- Búsqueda de una solución: Elección de los menos restringidos (heurística de minimización de conflictos.)

- Búsqueda de una solución óptima: Elección del mejor localmente según el criterio a optimizar.
- Búsqueda de todas las soluciones: Orden indiferente.

Elección de variables:

Orden Estático: Establecido antes de comenzar la búsqueda.
Basado en el criterio de estructura del grafo:

- La variable unida al más grande número de restricciones.
- La variable unida a las restricciones más difíciles.
- El orden de ancho máximo.

Orden Dinámico: Cambia con el transcurso de la búsqueda.
Solución adoptada más frecuentemente:

- Orden dinámico del más pequeño dominio.
- Filtro por forward checking o look-ahead.

Ejemplo: Elección de variables

Restricciones:

$$\begin{aligned} i &= k \\ j &= k \\ D_i &= \{a, b\} \\ D_j &= \{a, b\} \\ D_k &= \{a, b, c\} \end{aligned}$$

Orden i-j-k

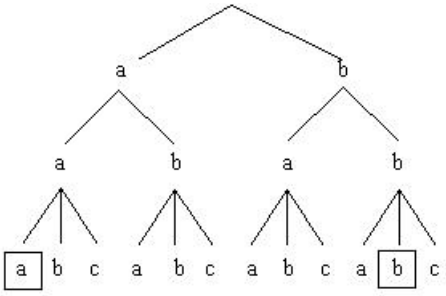


Figura 4.17: Orden i-j-k

Orden k-j-i

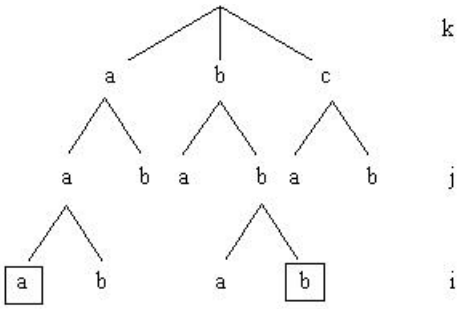


Figura 4.18: Orden k-j-i

Capítulo 5

Técnicas Incompletas

5.1. Búsqueda Alternativa

Heurística: Son criterios, principios o métodos que permite determinar entre un conjunto de posibilidades, aquella que promete ser la más eficaz para cumplir un objetivo. Las heurísticas representan el compromiso entre dos exigencias: la necesidad de tener criterios simples y al mismo tiempo la necesidad de distinguir entre una buena y una mala elección. Un método heurístico puede ser un método empírico utilizado para guiar acciones. Son algoritmos que buscan encontrar soluciones aceptables. Se usan porque ellas son, en general, eficientes computacionalmente y/o fáciles de implementar. Ellas nos son muy precisas ni predecibles. El mejor resultado se obtiene al mezclar heurísticas con técnicas o algoritmos de optimización.

Reparar una solución: Buscar una nueva solución a partir de una solución existente cambiando el valor que tiene asignado una sola variable. Es decir, se está buscando una nueva solución en el ”vecindario”de la solución actual.

Función Objetivo: Es la función clave dentro de la búsqueda. Debe reflejar el Objetivo que se está buscando y debe ser capaz de medir que tan buena (o mala) es una solución en términos del mismo. Una función objetivo es capaz de representar uno o varios objetivos, esto se logra a través de una ponderación de los ”subobjetivos”, la que es directamente proporcional al grado de importancia de ellos en la búsqueda.

Característica principal: Problemas NP-Completos, Explosión Combinatoria.

5.1.1. TSP (Travel Salesman Problem- Problema del Vendedor Viajero)

Problema:

Una persona debe visitar un conjunto de n ciudades comenzando en una ciudad específica, debiendo regresar a esa misma ciudad, luego de haber visitado todas las ciudades y sin haberse devuelto a ninguna.

Objetivo:

Encontrar la secuencia óptima de visita, medida a través de un criterio como: minimizando costo, minimizando tiempo o maximizando velocidad....

5.1.2. SPP (Space Planning problem - Problema de distribución espacial)

Problema:

Se tiene una pieza (un plano, una planta de un edificio, una placa) en donde se desean posicionar objetos sin que se superpongan.

El problema puede tener objetivos de al menos dos clases:

Optimización: Buscando introducir la mayor cantidad de objetos posibles en la pieza o buscar la menor dimensión de la pieza que permita que todos los objetos sean ubicados dentro de la pieza.

Satisfacción: Buscando la ubicación de los objetos que se sabe caben en la pieza.

5.1.3. TT (TimeTabling problem - Problema de distribución horaria)

Problema:

Una tabla horaria es la localización de un conjunto de reuniones en el tiempo. Una reunión es una combinación de recursos (salas, personas, equipos..) algunos de los cuales están especificados en el problema y otros se asignan como parte de la resolución del problema.

Puede tener básicamente 2 objetivos distintos:

Optimización: Minimizar el tiempo ocioso.

Satisfacción: Alcanzar a realizar todas las actividades en el tiempo dado.

5.1.4. k-Colouring Problem (Colorear grafos con k colores)

Problema:

Colorear un grafo con k colores de tal forma que los nodos adyacentes no tengan el mismo color.

Puede también tener 2 prismas en cuanto a objetivos:

Optimización: Encontrar el valor mínimo de k que permita cumplir con las restricciones.

Satisfacción: Dado un cierto valor k, por ejemplo $k = 3$, colorear el grafo con esos tres colores sin que los nodos adyacentes tengan el mismo color.

Podemos ver que estamos trabajando, en general, con problemas denominados ”Problemas de Satisfacción de Restricciones”(CSP) y ”Problemas de Optimización con Satisfacción de Restricciones”(CSOP) que pertenecen a la clase NP-completos debido a su complejidad.

5.2. Técnicas que usan heurísticas

5.2.1. Hill-Climbing (Escalando)

Corresponde a un término genérico para describir un algoritmo que busca ir mejorando el valor de la función objetivo de una casi-solución, al realizar reparaciones de la misma.

Algoritmos tipo hill-climbing, en general, comienzan con una casi-solución generada en forma aleatoria (asignándoles en forma aleatoria valores a las variables desde su dominio). Luego, realiza reparaciones de esa casi-solución buscando que la función objetivo de la nueva casi-solución sea mejor que la casi-solución actual.

Para ello es necesario contar con:

- Una función objetivo que mida la casi-solución.
- Un criterio para seleccionar la variable que se va a modificar.
- Un criterio para elegir un valor para esa variable seleccionada.

En particular para los problemas de satisfacción de restricciones (CSP), Minton et al. propusieron el siguiente algoritmo iterativo estocástico denominado *min-conflicts: Comienza con una casi-solución generada en forma completamente aleatoria y utiliza como*

- Función objetivo: Número de restricciones satisfechas
- Criterio para seleccionar la variable a modificar: Aleatorio
- Criterio para elegir un valor para la variable: Cambia el valor de la variable ssi existe otro valor dentro del dominio de la misma, que haga satisfacer más restricciones que la casi-solución actual.

Dificultad de *min-conflicts*: Se puede quedar en un óptimo local porque no hay mejores soluciones en el vecindario de la casi-solución actual.

Ejemplo:Heurísticas K-opt para búsqueda local TSP

Heurística K-Opt de Lin, Bell Systems Technical J.44 (1965) 2245-2269
Lin & Kernighan, Operations Research 21 (1973) 498-512

Definición K-exchange: Es un procedimiento que reemplaza K arcos de un tour de TSP por K nuevos arcos, tal que el tour resultante es un tour factible del TSP.

Ejemplo:

1. En el tour: A - B - C - D - E - A
   Reemplace (A, B) & (C, D) por (A , C) & (B, D) tour con “2-exchange”: A - C - B - D - A

2. En el tour para un TSP simétrico: A - B - C - D - E - A
   Reemplace (A, B), (C, D), & (D, E) por (B, D), (A, D), & (C, E) tour con “3-exchange”: A - D - B - C - E - A

Heurística K-Opt:
Sea N(S) el vecindario de una solución S formado por el conjunto de todas las soluciones que se pueden obtener a partir de ejecutar el procedimiento K-exchange sobre S.

Paso 1: Encontrar un tour inicial para el TSP usando cualquier heurística.
Esta es la primera solución S.

Paso 2: Buscar en el vecindario de S, N(S) hasta encontrar una solución S’ con F(S’) $<$ F(S), donde F(.) es la función objetivo a minimizar.
Reemplace S por S’.

Paso 3: Repetir el Paso 2 hasta que no se pueda encontrar una mejor solución en el vecindario N(S).

5.2.2. Tabu Search

Tabu search enfrenta el problema de ciclos impidiendo temporalmente movimientos que podrían hacer volver a una solución que ha sido recientemente visitada.

El efecto es prevenir ciclos a “corta” duración, sin embargo se pueden revisitar soluciones sobre grandes intervalos de tiempo.

Algoritmo Tabu Search:

Paso 0: Inicialización

X:=solución inicial factible

tmax:= máximo número de iteraciones

Mejor solución:=X

número de soluciones = t :=0

lista tabu:= vacía

Paso 1: Parada

Si cualquier movimiento posible de la solución actual es tabu o si t=tmax entonces parar. Entregar Mejor solución.

Paso 2: Mover

Elegir algún movimiento no-tabu factible $\triangle x(t+1)$

Paso 3: Iteración

Modificar $X(t+1) := X(t) + \triangle x(t+1)$

Paso 4: Reemplazar el mejor

Seleccionar aquel $X(t+1)$ dentro del vecindario con el mejor valor de la función objetivo Mejor solución:= X(t+1)

Paso 5: Actualizar Lista Tabu

Eliminar desde la lista tabu cualquier movimiento que ha permanecido un suficiente número de iteraciones en la lista.

Agregar un conjunto de movimientos que involucran un retorno inmediato desde X(t+1) a X(t)

Paso 6: Incrementar

t:=t+1, volver a Paso 1.

5.2.3. Aplicando Tabu Search al problema de la mochila

Problema de la mochila:

$$\text{máx } 18X_1 + 25X_2 + 11X_3 + 14X_4$$

s.a.

$$\begin{aligned} 2X_1 + 2X_2 + X_3 + X_4 &\leq 3 \\ X_1, \ldots, X_4 &= 0 \text{ ó } 1 \end{aligned}$$

Supuestos: Movimiento: Cambios de un componente 0 a 1 ó un componente 1 a 0. Solución inicial: (1,0,0,0)

1. Hill-Climbing

Vecinos permitidos:(0,0,0,0),(1,0,1,0) y (1,0,0,1) cuyas funciones de evaluación serían: 0, 29 y 32.

El que más mejora la función objetivo es (1,0,0,1).

Vecinos permitidos: (0,0,0,1) y (1,0,0,0) donde ninguno mejora la función objetivo entonces el óptimo encontrado sería 32.

2. Hill-Climbing con Restart

Nuevas soluciones iniciales: (0,1,0,0) y (0,0,1,0) óptimos serían 36 y 39.

3. Tabu con Sol. inicial: (1,0,0,0) con tmax:= 5 iteraciones

t	X(t)	Valor	Mejor Solución	Movimiento	△ Objetivo
0	(1,0,0,0)	18	18	j=4	14
1	(1,0,0,**1**)	32	32	j=1	-18
2	(**0**,0,0,1)	14	32	j=2	25
3	(**0**,**1**,1,1)	39	39	j=4	-14
4	(0,**1**,0,**0**)	25	39	j=3	11
5	(0,1,**1**,**0**)	36	39	parar	

5.2.4. Simulated Annealing

Simulated Annealing controla los ciclos aceptando movimientos que empeoran de acuerdo a una probabilidad comparada con el valor de un número generado aleatoriamente.

El movimiento será aceptado siempre que el movimiento mejore la función objetivo o en el caso que:

Probabilidad de aceptación = $\exp(\triangle obj/q)$

Algoritmo Simulated Annealing

Paso 0: Inicialización

X:=solución inicial factible

tmax:= máximo número de iteraciones

q:= Temperatura alta inicial

Mejor solución:= X

número de soluciones =t :=0

Paso 1: Parada

Si no hay un movimiento posible de la solución actual o si t=tmax entonces parar. Entregar Mejor solución.

Paso 2: Posible Movimiento

Elegir aleatoriamente algún movimiento factible $\triangle x(t+1)$

Calcular el incremento (o disminución) $\triangle$objetivo

Paso 3: Aceptación

Si X(t+1) mejora el objetivo o si $P(\exp(\triangle objetivo/q)) \Rightarrow random(0,1)$.

Modificar $X(t+1) := X(t) + \triangle x(t+1)$ sino volver al Paso 2.

Paso 4: Reemplazar el mejor

Si el valor de la función objetivo de X(t+1) es superior a Mejor solución entonces a Mejor solución:= X(t+1)

Paso 5: reducción de la Temperatura

Si ha pasado un número suficiente de iteraciones desde el último cambio de la temperatura, reduzca q.

Paso 6: Incrementar

t:=t+1, volver a Paso 1.

5.2.5. Aplicando el problema de la Mochila a Simulated Annealing

Solución inicial:= (1,0,0,0)
Temperatura inicial = q :=10
tmax:= 3
Números aleatorios generados: 0.72;0.83;0.33;0.41;0.09;0.94.

t	X(t)	Valor	Mejor Solución	q	Movimiento	△ Objetivo	Decisión
0	(1,0,0,0)	18	18	10	j=4	14	aceptado
1	(1,0,0,1)	32	32	10	j=4	-14	rechazado
					j=1	-18	aceptado
2	(0,0,0,1)	14	32	10	j=3	11	aceptado
3	(0,0,1,1)	25	32	10	parar		

Capítulo 6

Algoritmos Genéticos

6.1. Principales áreas de los A.E.

- Programación Evolucionista $rightarrow$ L. Fogel 1962 (San Diego, CA)
- Algorítmos Genéticos $rightarrow$ J. Holland 1962 (Ann Arbor, MI)
- Estrategias Evolucionistas $rightarrow$ I. Rechenberg & H.P. Schwefel 1965 (Berlin, Germany)
- Programación Genética $rightarrow$ J. Koza 1989 (Palo Alto, cA)

6.2. Introducción

La teoría de la evolución muestra que los seres vivos evolucionan bajo el efecto del medio ambiente: los que están mejor adaptados tienen más posibilidades de sobrevivir y de reproducirse. En cada generación las características de los individuos mejor adaptados tienen más posibilidades de estar presente en la población. La genética cuyo objetivo es estudiar los mecanismos de herencia propone un modelo que permite explicar la transmisin de estas características de una generación a otra.
Existen entidades responsables de la producción de caracteres hereditarios, que se llaman *genes* y el conjunto de genes de un individuo define su *genotipo*. Por otro lado, el *fenotipo* de un individuo corresponde a su apariencia física, más exactamente a un conjunto de caracter
isticas que uno puede observar, medir o calificar en él; el fenotipo es posible que cambie en el tiempo por efecto del medio en el cual evoluciona, pero sus variaciones no pueden transmitirse. Un gen es un segmento de *cromosoma*, parte del ADN que es el material genético de todas las células. Dos mecanismos permiten fabricar nuevas células. El primero es por división celular: una célula duplica su material genético antes de cortarse en dos copias de si misma. O casi copias fieles: un error de reproducción puede producirse, afectando un gen, ocurre entonces una *mutación* de ese gen. Este mecanismo lleva a cabo la reproducción *asexuada*. El segundo mecanismo hace intrevenir dos

padres para fabricar un hijo, es lo que se conoce como reproducción . En este mecanismo las células sexuales transmitidas por los padres aportan cada una la mitad de los cromosomas del hijo.
Los algoritmos genéticos están inspirados en estos mecanismos.

6.3. Estructura de un Programa Evolucionista

Inicio t:=0

inicializar P(t)

evaluar P(t)

Mientras (no sea condición de término) **haga**

t:=t+1

Seleccionar desde P(t-1)

Transformar usando operadores

Evaluar P(t)

Fin

Problema

$$\text{máx } f(X_1, X_2) = 21{,}5 + X_1 \sin\{4\Pi X_1\} + X_2 \sin\{20\Pi X_2\}$$

donde

$$\begin{aligned} -3{,}0 \leq \quad X_1 \quad &\leq 12{,}1 \\ 4{,}1 \leq \quad X_2 \quad &\leq 5{,}8 \end{aligned}$$

$\hookrightarrow$ Standard GA

6.4. Representación en un GA

- $X_1 \in [-3{,}0, 12{,}1]$. Este rango se divide en $15{,}1 \times 10000$ rangos de igual tamaño. En consecuencia se requiere 18 bits para representar la variable X_1.
- $X_2 \in [4{,}1, 5{,}8]$. Se divide en $1{,}7 \times 10000$ rangos de igual tamaño. Luego se requiere 15 bits para X_2.
- En total se requiere un string con 33 bits:
(010001001011010000111110010101000 10)

6.5. Conversión y Evaluación

- Si $X \in [a, b]$
- String de largo k que representa X: $(i_1, \ldots, i_k)$.
- El valor de X corresponde a: $a + decimal(i_1, \ldots, i_k) \times (b - a)/(2^k - 1)$.
- (01000100101101000011111001010100010).
- $decimal(i_1, \ldots, i_k) = \sum i_{k-j} \times 2^j, j = 0 \ldots k - 1$
- $X_1 = -3{,}0 + 70352 \times (15{,}1)/262143 = 1{,}0524$
- $X_2 = 5{,}7553$
- Función Objetivo (X_1, X_2)=20.2526

6.5.1. Ejemplo de Operadores

El problema es maximizar $f(x) = x^2$

Número	String	Aptitud	% del Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Roulette Wheel Selection

$$P = f_1 / \sum_{j-1}^{d} f_j$$

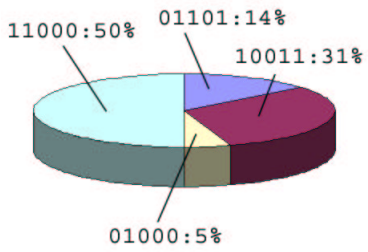


Figura 6.1: Ruleta

One-Point Crossover (SPX)

$$P_c \in [0{,}6 \ldots 1{,}0]$$

Padres:

- 01 | 101 (169)
- 11 | 000 (576)

Descendientes:

- 01000 (64)
- 11101 (841)

Mutación

$$P_m \in [0{,}001 \ldots 0{,}1]$$

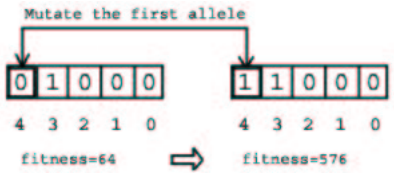


Figura 6.2: Mutación

6.6. Algoritmo Genético Standard

Este es una implementación de un algoritmo genético simple, donde la función de evaluación toma valores positivos solamente y la aptitud de un individuo es el mismo valor de la función objetivo.

```
#include <stdio.h> #include <stdlib.h> #include <math.h>

/* Change any of these parameters to match your needs */

#define POPSIZE 20              /* population size */ #define
MAXGENS 1000            /* max. number of generations */ #define
NVARS 2                 /* no. of problem variables */ #define
PXOVER 0.25             /* probability of crossover */ #define
PMUTATION 0.01          /* probability of mutation */ #define
TRUE 1 #define FALSE 0 #define PI 3.14159

int generation;                 /* current generation no. */ int
cur_best;                   /* best individual */ FILE *galog;
/* an output file */
```

```
struct genotype /* genotype (GT), a member of the population */ {
  double gene[NVARS];        /* a string of variables */
  double fitness;            /* GT's fitness */
  double upper[NVARS];       /* GT's variables upper bound */
  double lower[NVARS];       /* GT's variables lower bound */
  double rfitness;           /* relative fitness */
  double cfitness;           /* cumulative fitness */
};

struct genotype population[POPSIZE+1];    /* population */ struct
genotype newpopulation[POPSIZE+1]; /* new population; */
                                          /* replaces the */
                                          /* old generation */

/* Declaration of procedures used by this genetic algorithm */

void initialize(void); double randval(double, double); void
evaluate(void); void keep_the_best(void); void elitist(void); void
select(void); void crossover(void); void Xover(int,int); void
swap(double *, double *); void mutate(void); void report(void);
```

6.6.1. Función de inicialización

Inicializa los valores de los genes dentro de los límites de las variables. También inicializa (en cero) todos los valores de aptitud (*fitness*) para cada miembro de la población. La función lee los límites superiores e inferiores de cada variable desde un archivo de entrada '*gadata.txt*', y genera aleatoriamente valores entre estos límites para cada gen de cada genotipo en la población. El formato del archivo de entrada 'gadata.txt' es:

```
var1_lower_bound var1_upper bound\\
var2_lower_bound var2_upper bound\\


void initialize(void) { FILE *infile; int i, j; double lbound,
ubound;

if ((infile = fopen("gadata.txt","r"))==NULL)
      {
      fprintf(galog,"\nCannot open input file!\n");
      exit(1);
      }

/* initialize variables within the bounds */
```

```
for (i = 0; i < NVARS; i++)
      {
      fscanf(infile, "%lf",&lbound);
      fscanf(infile, "%lf",&ubound);

      for (j = 0; j < POPSIZE; j++)
           {
           population[j].fitness = 0;
           population[j].rfitness = 0;
           population[j].cfitness = 0;
           population[j].lower[i] = lbound;
           population[j].upper[i]= ubound;
           population[j].gene[i] = randval(population[j].lower[i],
                                    population[j].upper[i]);
           }
      }

fclose(infile); }

/*************************************************************/ /*
Random value generator: Generates a value within bounds */
/*************************************************************/

double randval(double low, double high) { double val;
val = ((double)(rand()%1000)/1000.0)*(high - low) + low;
return(val); }
```

6.6.2. Función de evaluación

Esta función toma una función definida por el usuario. Cada vez que es cambiada, el código debe ser recompilado.

La función dada en el ejemplo es:

$$21{,}5 + x[1] \times \sin\{4 \times PI \times x[1]\} + x[2] \times \sin\{20 \times PI \times x[2]\}$$

```
void evaluate(void) { int mem; int i; double x[NVARS+1];

for (mem = 0; mem < POPSIZE; mem++)
      {
      for (i = 0; i < NVARS; i++)
            x[i+1] = population[mem].gene[i];

      population[mem].fitness =
      21.5 + x[1]*sin(4*PI*x[1]) + x[2]*sin(20*PI*x[2]);
      }
```

```
}

/*************************************************************/
/* Keep_the_best function: This function keeps track of the    */
/* best member of the population. Note that the last entry in  */
/* the array Population holds a copy of the best individual    */
/*************************************************************/

void keep_the_best() { int mem; int i; cur_best = 0; /* stores the
index of the best individual */

for (mem = 0; mem < POPSIZE; mem++)
      {
      if (population[mem].fitness > population[POPSIZE].fitness)
            {
            cur_best = mem;
            population[POPSIZE].fitness = population[mem].fitness;
            }
      }
/* once the best member in the population is found, copy the genes
*/ for (i = 0; i < NVARS; i++)
      population[POPSIZE].gene[i] = population[cur_best].gene[i];
}
```

6.6.3. Función elitista

El mejor miembro de la generación previa es almacenada como el último en el arreglo. Si el mejor miembro de la generación actual es peor que el mejor miembro de la generación anterior, este último reemplazaría al peor miembro de la población actual.

```
void elitist() { int i; double best, worst;             /* best
and worst fitness values */ int best_mem, worst_mem; /* indexes of
the best and worst member */

best = population[0].fitness; worst = population[0].fitness; for
(i = 0; i < POPSIZE - 1; ++i)
      {
      if(population[i].fitness > population[i+1].fitness)
            {
            if (population[i].fitness >= best)
                  {
                  best = population[i].fitness;
                  best_mem = i;
                  }
            if (population[i+1].fitness <= worst)
```

```
                {
                worst = population[i+1].fitness;
                worst_mem = i + 1;
                }
          }
      else
          {
          if (population[i].fitness <= worst)
                {
                worst = population[i].fitness;
                worst_mem = i;
                }
          if (population[i+1].fitness >= best)
                {
                best = population[i+1].fitness;
                best_mem = i + 1;
                }
          }
      }
/* if best individual from the new population is better than */ /*
the best individual from the previous population, then    */ /*
copy the best from the new population; else replace the   */ /*
worst individual from the current population with the     */ /*
best one from the previous generation                     */

if (best >= population[POPSIZE].fitness)
    {
    for (i = 0; i < NVARS; i++)
       population[POPSIZE].gene[i] = population[best_mem].gene[i];
    population[POPSIZE].fitness = population[best_mem].fitness;
    }
else
    {
    for (i = 0; i < NVARS; i++)
       population[worst_mem].gene[i] = population[POPSIZE].gene[i];
    population[worst_mem].fitness = population[POPSIZE].fitness;
    }
}
```

6.6.4. Función de selección

La selección proporcional estándar para los problemas de maximización incorporan el modelo elitista -asegurarse que el mejor miembro sobreviva.

```
void select(void) { int mem, i, j, k; double sum = 0; double p;
```

```
/* find total fitness of the population */ for (mem = 0; mem <
POPSIZE; mem++)
      {
      sum += population[mem].fitness;
      }

/* calculate relative fitness */ for (mem = 0; mem < POPSIZE;
mem++)
      {
      population[mem].rfitness =  population[mem].fitness/sum;
      }
population[0].cfitness = population[0].rfitness;

/* calculate cumulative fitness */ for (mem = 1; mem < POPSIZE;
mem++)
      {
      population[mem].cfitness =  population[mem-1].cfitness +
                        population[mem].rfitness;
      }

/* finally select survivors using cumulative fitness. */

for (i = 0; i < POPSIZE; i++)
      {
      p = rand()%1000/1000.0;
      if (p < population[0].cfitness)
            newpopulation[i] = population[0];
      else
            {
            for (j = 0; j < POPSIZE;j++)
                  if (p >= population[j].cfitness &&
                              p<population[j+1].cfitness)
                        newpopulation[i] = population[j+1];
            }
      }
/* once a new population is created, copy it back */

for (i = 0; i < POPSIZE; i++)
      population[i] = newpopulation[i];
}
```

6.6.5. Selección crossover

Selecciona dos padres para realizar el crossover. Implementa un crossover de un solo punto.

```
void crossover(void) { int i, mem, one; int first  =  0; /* count
of the number of members chosen */ double x;

for (mem = 0; mem < POPSIZE; ++mem)
      {
      x = rand()%1000/1000.0;
      if (x < PXOVER)
            {
            ++first;
            if (first % 2 == 0)
                  Xover(one, mem);
            else
                  one = mem;
            }
      }
}
```

6.6.6. Crossover

Implementa un crossover de dos padres seleccionados.

```
void Xover(int one, int two) { int i; int point; /* crossover
point */

/* select crossover point */ if(NVARS > 1)
   {
   if(NVARS == 2)
         point = 1;
   else
         point = (rand() % (NVARS - 1)) + 1;

   for (i = 0; i < point; i++)
        swap(&population[one].gene[i], &population[two].gene[i]);

   }
}
```

6.6.7. Swap

Un procedimiento de swap que ayuda en el swapping de dos variables.

```
void swap(double *x, double *y) { double temp; temp = *x; *x = *y;
*y = temp; }
```

6.6.8. Mutación

Mutación aleatoria uniforme. Una variable seleccionada para mutación es reemplazada por un valor aleatorio entre los límites inferior y superior para esa variable.

```
void mutate(void) { int i, j; double lbound, hbound; double x;

for (i = 0; i < POPSIZE; i++)
      for (j = 0; j < NVARS; j++)
            {
            x = rand()%1000/1000.0;
            if (x < PMUTATION)
                  {
                  /* find the bounds on the variable to be mutated */
                  lbound = population[i].lower[j];
                  hbound = population[i].upper[j];
                  population[i].gene[j] = randval(lbound, hbound);
                  }
            }
}
```

6.6.9. Función de reporte

Reporta el progreso de la simulación. Datos descargados en el archivo de salida son separados por comas.

```
void report(void) { int i; double best_val;           /* best
population fitness */ double avg;                /* avg
population fitness */ double stddev;             /* std.
deviation of population fitness */ double sum_square;          /*
sum of square for std. calc */ double square_sum;          /*
square of sum for std. calc */ double sum;                 /*
total population fitness */

sum = 0.0; sum_square = 0.0;

for (i = 0; i < POPSIZE; i++)
      {
      sum += population[i].fitness;
      sum_square += population[i].fitness * population[i].fitness;
      }

avg = sum/(double)POPSIZE; square_sum = avg * avg * POPSIZE;
stddev = sqrt((sum_square - square_sum)/(POPSIZE - 1)); best_val =
population[POPSIZE].fitness;
```

```
fprintf(galog, "\n%5d,      %6.3f, %6.3f, %6.3f \n\n", generation,
                                    best_val, avg, stddev);
}
```

6.6.10. Función principal (main)

Cada generación implica seleccionar el mejor miembro, aplicar crossover y mutación, luego evaluar la población resultante, hasta que la condición de término es satisfecha.

```
void main(void) { int i;

if ((galog = fopen("galog.txt","w"))==NULL)
      {
      exit(1);
      }
generation = 0;

fprintf(galog, "\n generation  best  average  standard \n");
fprintf(galog, " number      value fitness  deviation \n");

initialize(); evaluate(); keep_the_best();
while(generation<MAXGENS)
      {
      generation++;
      select();
      crossover();
      mutate();
      report();
      evaluate();
      elitist();
      }
fprintf(galog,"\n\n Simulation completed\n"); fprintf(galog,"\n
Best member: \n");

for (i = 0; i < NVARS; i++)
   {
   fprintf (galog,"\n var(%d) = %3.3f",i,population[POPSIZE].gene[i]);
   }
fprintf(galog,"\n\n Best fitness = %3.3f",population[POPSIZE].fitness);
fclose(galog); printf("Success\n"); }
```

6.7. Operadores No-standard

Figura 6.3: Operadores No-standard

6.8. Ideas: Operadores de Recombinación

1. Two-points crossover. Dos padres → dos hijos
2. Uniform crossover: Dos padres → dos hijos Selección aleatoria bit por bit.
3. (n,p,g): Un padre → un hijo
   - n: número de variables a modificar $\{\sharp \ldots\}$
   - p: criterio de selección de las posiciones a modificar {r,b}
   - g: criterio de selección de los nuevos valores {r,b}
4. Knowledge-augmented crossover. Dos padres → un hijo
   - Si las variables no están en conflicto en *ambos* padres elegir uno al hazar
   - Si hay un conflicto en un sólo padre, heredar del padre que no está en conflicto
   - Si los dos padres están en conflicto elegir el menos conflictivo de los dos.
5. UAX: Dos padres → un hijo

   Procedure UAX (Parent-1,Suppl-1, parent-2, suppl-2)
   Begin
   if r¡pcross **then**
   parent-select=selection(Parent-1,parent-2)
   parent-not-selected = (Parent-1, parent-2) - (*parent-select*)
   for i:=1 to n **do**
   if $Suppl - parent - select$[i] = suppl-parent-not-selecte[i] **then**

interchange parents
end if
child[i]=$parent - select$[i]
suppl-child[i]=$suppl - parent - select$[i]
end for
end if

End

6.9. Representación Ordinal

- Lista de referencia (1 2 3 4 5 6 7 8 9)
- Un tour: 1 - 2 - 4 - 3 - 8 - 5 - 9 - 7
- Lista: (1 1 2 1 4 3 1 1)
- Cruzar dos padres:
  P1: (1 1 2 1 | 4 1 3 1 1)
  P2: (5 1 5 5 | 5 3 3 2 1)

  $Rightarrow$
  1 - 2 - 4 - 3 - 8 -5 - 9 - 6 - 7
  5 - 1 - 7 - 8 - 9 - 4 - 6 - 3 - 2

  1 - 2 - 4 - 3 - 9 - 7 - 8 - 6 - 5
  5 - 1 - 7 - 8 - 6 - 2 - 9 - 3 - 4

6.10. Problema del Vendedor Viajero

Aspectos Representación.

1. Tour: 5-1-7-8-9-4-6-2-3
   Representación:(5 1 7 8 9 4 6 2 3)

   Operador: PMX (Partially Mapped Crossover)

   P1:(1 2 3 | 4 5 6 7 | 8 9)
   P2:(4 5 2 | 1 8 7 6 | 9 3)

   a) (X X X | 1 8 7 6 | X X)
   (X X X | 4 5 6 7 | X X)

   b) Sin conflictos
   (X 2 3 | 1 8 7 6 | X 9)
   (X X 2 | 4 5 6 7 | 9 3)

c) Cambios
(4 2 3 | 1 8 7 6 | 5 9)
(1 8 2 | 4 5 6 7 | 9 3)

2. Lista referencia (1 2 3 4 5 6 7 8 9)
Tour: 1-2-4-3-8-5-9-6-7
Representación: (1 1 2 1 4 1 3 1 1)

P1:(1 1 2 1 | 4 1 3 1 1)
P2:(5 1 5 5 | 5 3 3 2 1)

a) Padres
(1 2 4 3 8 5 9 6 7)
(5 1 7 8 9 4 6 3 2)

b) Hijos
(1 2 4 3 9 7 8 6 5)
(5 1 7 8 6 2 9 3 4)

6.11. Problema

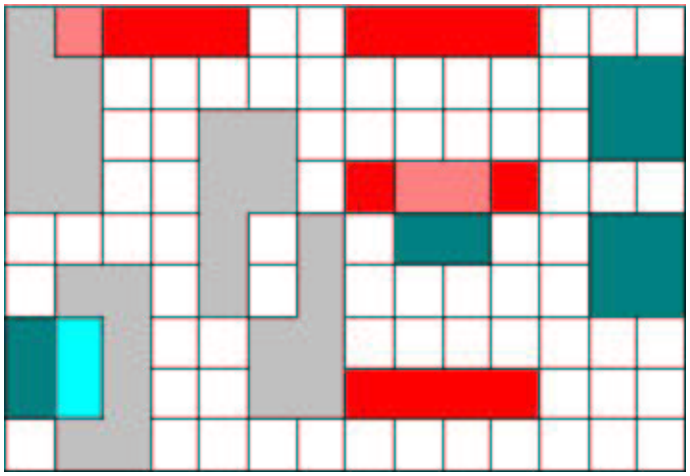


Figura 6.4: Spaceplaning

Características:

- Posee una cantidad finita de objetos (no es un problema de optimización)
- Diferentes dimensiones de objetos (sólo rectángulos)

Restricciones:

- No se permiten rotaciones
- Las dimensiones de un objeto no pueden salir del espacio delimitado
- No se permiten superposiciones entre los objetos