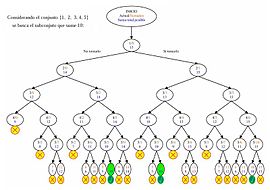
Vuelta atrás

Vuelta atrás, (Backtracking) es una estrategia para encontrar soluciones a problemas que satisfacen restricciones. El término "backtrack" fue acuñado por primera vez por el matemático estadounidense [D. H. Lehmer](http://es.wikipedia.org/w/index.php?title=Derrick_Henry_Lehmer&action=edit&redlink=1) en la [década de 1950](http://es.wikipedia.org/wiki/D%C3%A9cada_de_1950).

[](http://commons.wikimedia.org/wiki/File:Branch%26bound_low.jpg)

Ejemplo de árbol de búsqueda. Nótese que el árbol no tiene por qué ser binario.

Índice

  [[ocultar](http://es.wikipedia.org/wiki/Backtracking)]

* [1 Concepto](http://es.wikipedia.org/wiki/Backtracking#Concepto)
* [2 Enfoques](http://es.wikipedia.org/wiki/Backtracking#Enfoques)
* [3 Diseño e implementación](http://es.wikipedia.org/wiki/Backtracking#Dise.C3.B1o_e_implementaci.C3.B3n)
* [4 Heurísticas](http://es.wikipedia.org/wiki/Backtracking#Heur.C3.ADsticas)
* [5 Ejemplos de aplicación de backtracking](http://es.wikipedia.org/wiki/Backtracking#Ejemplos_de_aplicaci.C3.B3n_de_backtracking)
* [6 Ejemplos de problemas comunes resueltos usando Vuelta Atrás](http://es.wikipedia.org/wiki/Backtracking#Ejemplos_de_problemas_comunes_resueltos_usando_Vuelta_Atr.C3.A1s)
* [7 Aplicaciones](http://es.wikipedia.org/wiki/Backtracking#Aplicaciones)

[8 Branch & Bound (Ramificación y poda)](http://es.wikipedia.org/wiki/Backtracking#Branch_.26_Bound_.28Ramificaci.C3.B3n_y_poda.29)

Concepto

En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

Algoritmo de Backtracking

proc Backtracking (↕X[1 . . . i ]: TSolución, ↑ok: B)

variables L: ListaComponentes

inicio

si EsSolución (X) entonces ok CIERTO

en otro caso

ok FALSO

L=Candidatos (X)

mientras ¬ok ^ ¬Vacía (L) hacer

X[i + 1] Cabeza (L); L Resto (L)

Backtracking (X, ok)

finmientras

finsi

fin

Podemos visualizar el funcionamiento de una técnica de backtracking como la exploración en profundidad de un grafo.

Cada vértice del grafo es un posible estado de la solución del problema. Cada arco del grafo representa la transición entre dos estados de la solución (i.e., la toma de una decisión).

Típicamente el tamaño de este grafo será inmenso, por lo que no existirá de manera explícita. En cada momento sólo tenemos en una estructura los nodos que van desde el estado inicial al estado actual. Si cada secuencia de decisiones distinta da lugar a un estado diferente, el grafo es un árbol (el árbol de estados).

[[editar](http://es.wikipedia.org/w/index.php?title=Vuelta_atr%C3%A1s&action=edit&section=2)]Enfoques

Los problemas que deben satisfacer un determinado tipo de restricciones son problemas completos, donde el orden de los elementos de la solución no importa. Estos problemas consisten en un conjunto (o lista) de variables a la que a cada una se le debe asignar un valor sujeto a las restricciones del problema. La técnica va creando todas las posibles combinaciones de elementos para obtener una solución. Su principal virtud es que en la mayoría de las implementaciones se puede evitar combinaciones, estableciendo funciones de acotación (o poda) reduciendo el [tiempo](http://es.wikipedia.org/w/index.php?title=Eficiencia_de_los_algoritmos&action=edit&redlink=1) de ejecución.

Vuelta atrás está muy relacionado con la [búsqueda combinatoria](http://es.wikipedia.org/w/index.php?title=B%C3%BAsqueda_combinatoria&action=edit&redlink=1).

[[editar](http://es.wikipedia.org/w/index.php?title=Vuelta_atr%C3%A1s&action=edit&section=3)]Diseño e implementación

Esencialmente, la idea es encontrar la mejor combinación posible en un momento determinado, por eso, se dice que este tipo de algoritmo es una[búsqueda en profundidad](http://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad). Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción (hijo [si nos referimos a un árbol]). Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja).

Normalmente, se suele implementar este tipo de algoritmos como un procedimiento [recursivo](http://es.wikipedia.org/wiki/Recursividad). Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados. La diferencia con la[búsqueda en profundidad](http://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad) es que se suelen diseñar funciones de cota, de forma que no se generen algunos estados si no van a conducir a ninguna solución, o a una solución peor de la que ya se tiene. De esta forma se ahorra espacio en memoria y tiempo de ejecución.

[[editar](http://es.wikipedia.org/w/index.php?title=Vuelta_atr%C3%A1s&action=edit&section=4)]Heurísticas

Algunas heurísticas son comúnmente usadas para acelerar el proceso. Como las variables se pueden procesar en cualquier orden, generalmente es más eficiente intentar ser lo más restrictivo posible con las primeras (esto es, las primeras con menores valores posibles). Este proceso poda el [árbol de búsqueda](http://es.wikipedia.org/wiki/%C3%81rbol_(estructura_de_datos)) antes de que se tome la decisión y se llame a la subrutina recursiva.

Cuando se elige qué valor se va a asignar, muchas implementaciones hacen un examen hacia delante (FC, Forward Checking), para ver qué valor restringirá el menor número posible de valores, de forma que se anticipa en a) preservar una posible solución y b) hace que la solución encontrada no tenga restricciones destacadas.

Algunas implementaciones muy sofisticadas usan una función de cotas, que examina si es posible encontrar una solución a partir de una solución parcial. Además, se comprueba si la solución parcial que falla puede incrementar significativamente la eficiencia del algoritmo. Por el uso de estas funciones de cota, se debe ser muy minucioso en su implementación de forma que sean poco costosas computacionalmente hablando, ya que lo más normal es que se ejecuten en para cada nodo o paso del algoritmo. Cabe destacar, que las cotas eficaces se crean de forma parecida a las funciones [heurísticas](http://es.wikipedia.org/wiki/Heur%C3%ADstica), esto es, relajando las restricciones para conseguir mayor eficiencia.

Con el objetivo de mantener la solución actual con coste mínimo, los algoritmos vuelta atrás mantienen el coste de la mejor solución en una variable que va variando con cada nueva mejor solución encontrada. Así, si una solución es peor que la que se acaba de encontrar, el algoritmo no actualizará la solución. De esta forma, devolverá siempre la mejor solución que haya encontrado.

[[editar](http://es.wikipedia.org/w/index.php?title=Vuelta_atr%C3%A1s&action=edit&section=5)]Ejemplos de aplicación de backtracking

SATISFABILITY

Inicialmente A contiene la expresión booleana que constituye el problema.

Elegir subproblema de A, p.ejemplo : (x+y+z)(x'+y)(y'+z)(z'+x)(x'+y'+z').

Elegir una cláusula con mínimo número de literales.

Elegir una variable x, y, z,... dentro de la cláusula y crear 2 subproblemas reemplazando x=V y x=F.

En el caso x=V

Omitir las cláusulas donde aparece x.

Omitir x' en las cláusulas que aparece x'.

En el caso x=F

Omitir las cláusulas donde aparece x'.

Omitir x en las cláusulas que aparece x.

Test

Si no quedan cláusulas. STOP. (solución encontrada).

Si hay una cláusula vacía. DROP.

En otro caso añadir a A

Nota: Observemos que si encontramos a A vacío entonces la expresión booleana no puede ser satisfecha.

HAMILTON CYCLE (VIAJANTE DE COMERCIO)

En este caso los subproblemas S son caminos que parten de a y llegan a b a través de un sucesión de nodos T. (b es el mismo a lo largo de todo el algoritmo).

Inicialmente A contiene solamente el camino (a, vacío, b).

Elegimos un subproblema S cualquiera de A (y lo borramos de A) y añadimos ramas (c, a) del grafo (las c's son las adyacentes de a) . Estos caminos extendidos son los hijos. Ahora cada c juega el rol de a.

Examinamos c/ hijo:

Test:

1)Si G-T forma un camino hamiltoniano STOP (solución hallada)

2)Si G-T tiene un nodo de grado uno (excepto a y b) o si G-T-{a, b} es disconexo entonces DROP este subproblema .

3) Si 1) y 2) fallan add subproblema en A.

EXACT COVER

Dado un conjunto finito U y una familia se subconjuntos {Tj} de U definimos una matriz A donde cada fila se corresponde con un elemento ui de U y cada columna de A con un subconjunto Tj . Ponemos aij=1 si ui∈U pertenece a Tj y aij=0 en caso contrario. Interpretamos que xj=1 significa que elegimos Tj y 0 en caso contrario.

Se trata de averiguar si es factible Ax=1 donde A y x son binarias y las componentes de 1 son unos.

S0= un vector de ceros (raíz del árbol)

Cada nodo S del árbol es una sucesión x cuyas primeras k componentes le han sido asignados un 1 o un 0 y el resto de componentes son ceros. Reemplazamos S por 2 subproblemas Si (i=1,2) poniendo xk+1 =1 y xk+1=0 respectivamente.

Test

if Ax=1 STOP

if Ax>1 DROP Si

if Ax<1 add Si to A

Ejemplos de problemas comunes resueltos usando Vuelta Atrás

* [Sudoku](http://es.wikipedia.org/wiki/Sudoku_backtracking)
* [Problema de los movimientos de un caballo](http://es.wikipedia.org/wiki/Problema_de_los_movimientos_de_un_caballo)
* [Las ocho reinas](http://es.wikipedia.org/wiki/Las_ocho_reinas)

[[editar](http://es.wikipedia.org/w/index.php?title=Vuelta_atr%C3%A1s&action=edit&section=7)]Aplicaciones

Vuelta atrás se usa en la implementación de los [lenguajes de programación](http://es.wikipedia.org/wiki/Lenguajes_de_programaci%C3%B3n)tales como [Lenguaje de programación Planner](http://es.wikipedia.org/wiki/Planner) y [Prolog](http://es.wikipedia.org/wiki/Prolog" \o "Prolog). Además, se usa en los análisis sintácticos de los compiladores. Su uso en [inteligencia artificial](http://es.wikipedia.org/wiki/Inteligencia_artificial) ha sido muy importante, dando lugar a nuevos tipos de búsquedas como el [A estrella](http://es.wikipedia.org/wiki/A*).

[[editar](http://es.wikipedia.org/w/index.php?title=Vuelta_atr%C3%A1s&action=edit&section=8)]Branch & Bound (Ramificación y poda)

Este método busca una solución como en el método de backtracking, pero cada solución tiene asociado un costo y la solución que se busca es una de mínimo costo llamada óptima. Además de ramificar una solución padre (branch) en hijos se trata de eliminar de consideración aquellos hijos cuyos descendientes tienen un costo que supera al óptimo buscado acotando el costo de los descendientes del hijo (bound). La forma de acotar es un arte que depende de cada problema. La acotacion reduce el tiempo de búsqueda de la solución óptima al "podar" (pruning) los subarboles de descendientes costosos.