Tecnológico de Monterrey.

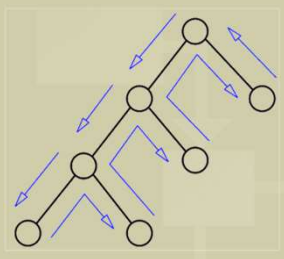
Campus Querétaro.

TC2038. Análisis y diseño de Algoritmos A

M.C. Ramona Fuentes Valdéz

rfuentes@tec.mx

49



Retroceso o vuelta atrás
(Backtracking)

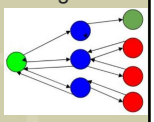
50

Técnicas de diseño de algoritmos

Algoritmos Backtracking

(Árbol del espacio de estados)

- El *retroceso* es un algoritmo general para encontrar todas (*o algunas*) soluciones a algunos problemas computacionales, en particular los problemas de satisfacción por restricciones.
  - Realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones.
  - Resuelve un problema de forma incremental.
- Genera gradualmente candidatos para las soluciones y abandona cada candidato parcial ("retrocede") tan pronto como determina que el candidato no puede llegar a una solución válida.
- Es una técnica aplicable a problemas de:
  - Decisión utilizado para encontrar una solución factible del problema.
  - Optimización utilizado para encontrar la mejor solución que se pueda aplicar.
  - Enumeración utilizado para encontrar el conjunto de todas las soluciones factibles del problema.
  - Juegos, etc.



- ***Resultado:*** puede resultar ineficiente para espacios de búsqueda muy grandes.
- Se puede ver como "*opuesto*" de la técnica *greedy*:

  Greedy: añade elementos a la solución y no deshace ninguna decisión.

  Backtracking: añade y quita elementos a la solución, probando todas las combinaciones posibles.

Técnicas de diseño de algoritmos

Algoritmos Backtracking

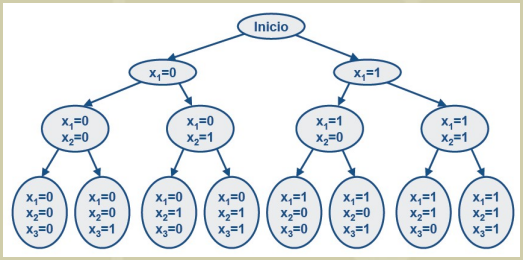
Funcionamiento

- Considera una solución como una tupla de acciones $(x_1,x_2,\ldots,x_n)$ que satisfaga unas restricciones.
- Suponiendo que ***k*** es un nivel que representa a todas las soluciones parciales de tamaño ***k*** $(x_1,x_2,\ldots,x_k)$:
  - Si se puede añadir un nuevo elemento a la solución x_{k+1}, se genera y avanza al nivel ***k+1.***
  - Si no, se prueban otros valores para x_k.
  - Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior k-1.
  - Se sigue hasta obtener una solución óptima a todas las soluciones posibles.

Algoritmo

```
Procedure 1 BACKTRACKING
Input: node
  if SOLUTION(node)) then
    print node
  end if
  if NOT(PROMISSORY(node))) then
    return
  end if
  for i ← 1 to CHILDS(node) do
    BACKTRACKING(i)
  end for
```

El resultado es equivalente a un recorrido en profundidad en un árbol de soluciones



Técnicas de diseño de algoritmos

Algoritmos Backtracking

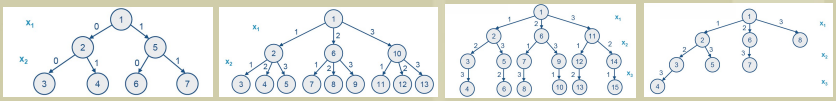
Árboles fantasma

El árbol en el backtracking:

- Es una representación del espacio de búsqueda del algoritmo
  - ➔ el árbol es implícito, no almacenado
- Su recorrido es en profundidad, normalmente de izquierda a derecha.

Algunos tipos comunes de árboles en el backtracking:

- Binarios
- *n*-arios
- Permutacionales
- Combinatorios



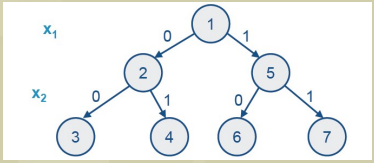
53

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Árboles binarios

Sea $s=(x_1,x_2,x_3,\ldots,x_n)$ con $x_i \in \{0,1\}$

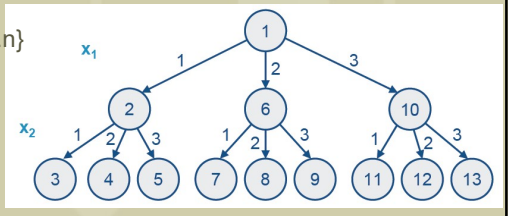


Tipo de problemas: elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos.

- Problema de la mochila.
- Encontrar un subconjunto de {12, 1, 31, 9, 11, 10} que sume exactamente 50.

Árboles *n*-arios

Sea $s=(x_1,x_2,x_3,\ldots,x_n)$ con $x_i \in \{1..n\}$



Tipo de problemas: varias opciones para un cada x_i

- Problema de cambio de monedas
- Problema de las *n* reinas

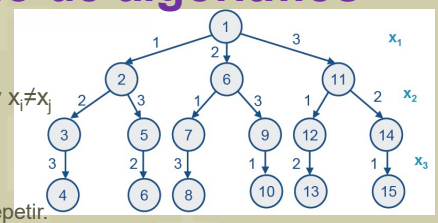
54

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Árboles permutacionales

Sea $s=(x_1,x_2,x_3,\ldots,x_n)$ con $x_i \in \{1,..,n\}$ y $x_i \neq x_j$

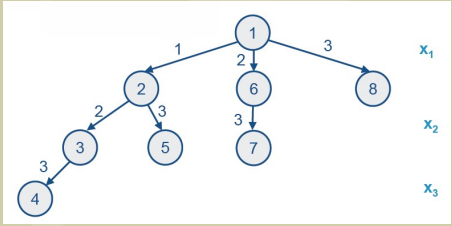


Tipo de problemas: los x_i no se pueden repetir.

- Generar todas las permutaciones de (1, ..., n).
- Asignar *n* trabajos a *n* personas, asignación uno-a-uno.

Árboles combinacionales

Sea Sea $s=(x_1,x_2,x_3,\ldots,x_m)$ con $m \leq n$, $x_i \in \{1,..,n\}$ y $x_i<x_i+1$



Tipo de problemas: elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos (*los mismos que con los binarios*).

- Problema de la mochila.
- Encontrar un subconjunto de {12, 1, 31, 9, 11, 10} que sume exactamente 50.

55

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio:

Dado un conjunto S de números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumpla, se debe responder con el conjunto vacío.

Si tenemos que: S= {2, 3, 7, 9} y C = 9, se desea encontrar cuál es el subconjunto cuyos elementos realizan esa suma.

Solución

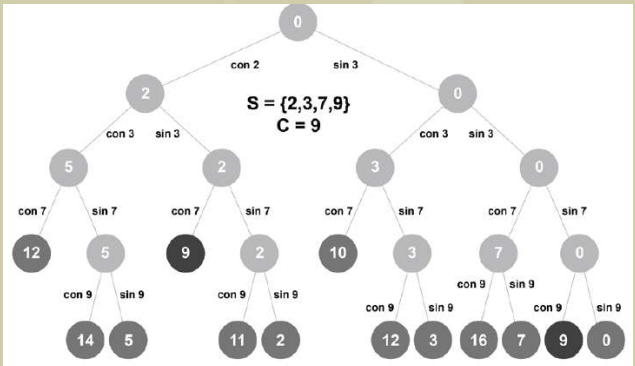
Fuerza bruta Tabla de verdad de 2^n renglones → Se forman todas las posibles sumas.

2	3	7	9	SUMA
F	F	F	F	NA
F	F	F	V	9
F	F	V	F	7
F	F	V	V	16
F	V	F	F	3
F	V	F	V	12
F	V	V	F	10
F	V	V	V	19
V	F	F	F	2
V	F	F	V	11
V	F	V	F	9
V	F	V	V	18
V	V	F	F	5
V	V	F	V	14
V	V	V	F	12
V	V	V	V	21

La solución es: {2, 7} y {9}.

Si C=25 no hay solución.
Respuesta = conjunto vacío {}.

Backtracking



56

Técnicas de diseño de algoritmos

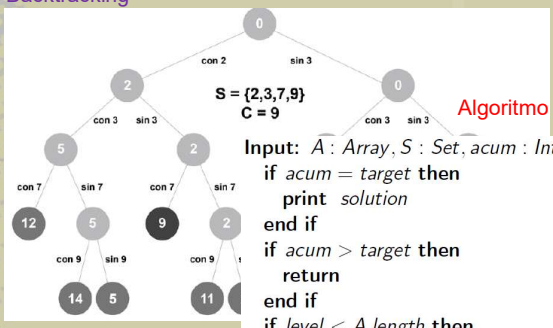
Algoritmos Backtracking

Ejercicio:

Dado un conjunto S de números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumpla, se debe responder con el conjunto vacío.

Si tenemos que: S= {2, 3, 7, 9} y C = 9, se desea encontrar cuál es el subconjunto cuyos elementos realizan esa suma.

Solución Backtracking



La solución es: {2, 7} y {9}.

Algoritmo

```
Input: A : Array, S : Set, acum : Integer, target : Integer, level : Integer
  if acum = target then
    print solution
  end if
  if acum > target then
    return
  end if
  if level < A.length then
    SUBSET_SUM(A, S, acum, target, level + 1)
    S.INSERT(A[level])
    SUBSET_SUM(A, S, acum + A[level], target, level + 1)
  end if
```


57

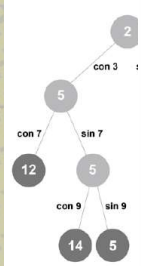
Técnicas de diseño de algoritmos

Algoritmos

Ejercicio:

Dado un conjun
la cantidad C. S
conjunto vacío.

Solución Backtracking



La solución es:

```cpp
#include <iostream>
#include <set>
#include <vector>

using namespace std;

vector<int> S{2,3,7,9};  // conjunto de números
int C = 9;  // suma deseada

void imprimeConjunto(set<int> s){
  cout << "{";
  for (int i : s)
    cout << i << " ";
  cout << "}\n";
}

void backtracking(set<int> s, int suma, int nivel, int c){
  if (suma > c) return; // se termina la exploración de esa rama
  if (suma == c){       // es una solución
    imprimeConjunto(s);
    return;
  }
  if (nivel < S.size()){  // todavía no se llega al final
    backtracking(s, suma, nivel+1, c);
    s.insert(S[nivel]);
    backtracking(s, suma+S[nivel], nivel+1, c);
  }
}

int main(){
  backtracking({}, 0, 0, C);
}
```


58

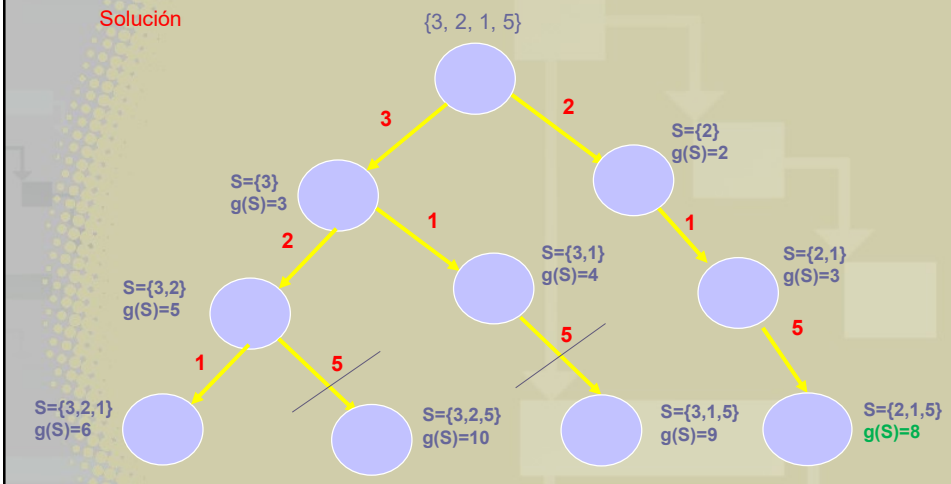
Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio:

Dado un conjunto de números enteros {3, 2, 1, 5}, encontrar si existe algún subconjunto de 3 elementos distintos cuya suma sea exactamente 8.

Solución



59

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio:

Hallar todas las permutaciones de un número *N*.

- Por ejemplo, las permutaciones de 123 son: 123, 231, 321, 312, 132, 213, 123.

Solución

Algoritmo

```
Input: S : String, pos : Integer
  if pos = 0 then
    print S
  else
    for i ← 1 to pos do
      SWAP(S, i, pos)
      PERMUTATION(S, pos − 1)
      SWAP(S, i, pos)
    end for
  end if
```


60

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Decisiones

- ¿Qué tipo de árbol es adecuado para el problema?
  - ¿Cómo es la representación de la solución?
  - ¿Cómo es la tupla solución?
  - ¿Qué indica cada x_i y qué valores puede tomar?
- ¿Cómo generar un recorrido según ese árbol?
  - Generar un nuevo nivel.
  - Generar los hermanos de un nivel.
  - Retroceder en el árbol.
- ¿Qué ramas se pueden descartar por no conducir a soluciones del problema?
  - Poda por restricciones del problema.
  - Poda según el criterio de la función objetivo.

61

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Esquema general (*no recursivo*)

Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad, y se supone que existe alguna.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s) // genera el siguiente hermano, o el primero, para el nivel actual
        si Solución (nivel, s) entonces  //comprueba si la tupla(s[1], ..., s[nivel]) es una solución válida
            fin:= true
        sino si Criterio (nivel, s) entonces
                //comprueba si a partir de (s[1], ..., s[nivel]) se puede alcanzar una solución válida.
                //En otro caso se rechazarán todos los descendientes (poda).
            nivel:= nivel + 1
        sino mientras NOT MasHermanos (nivel, s) hacer
                //devuelve true si hay más hermanos del nodo actual que todavía no han sido generados.
            Retroceder (nivel, s)
                // retrocede un nivel en el árbol de soluciones. Disminuye en 1 el valor de nivel, y
                // posiblemente tendrá que actualizar la solución actual, quitando los elementos retrocedidos.
    hasta fin
```

Es común que se utilicen variables que lleven un registro del valor asociado (beneficio, peso, etc.) a la secuencia de acciones de la tupla solución.

¿Qué pasa si…

1) no es seguro que exista una solución?
2) queremos almacenar todas las soluciones (*no sólo una*)?
3) el problema es de optimización (*maximizar o minimizar*)?

→ Variaciones

62

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Esquema general (*no recursivo*)

Caso 1) Puede que no exista ninguna solución.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
            Generar (nivel, s)
            si Solución (nivel, s) entonces
               fin:= true
            sino si Criterio (nivel, s) entonces
               nivel:= nivel + 1
            sino
               mientras NOT MasHermanos (nivel, s)
                    Retroceder (nivel, s)
            finsi
    hasta fin OR (nivel==0)
```

Para poder generar todo el árbol de *backtracking*

AND (nivel>0)

63

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Esquema general (*no recursivo*)

Caso 2) Queremos almacenar todas las soluciones.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
            Generar (nivel, s)
            si Solución (nivel, s) entonces
               Almacenar (nivel, s)
            si Criterio (nivel, s) entonces
               nivel:= nivel + 1
            sino
               mientras NOT MasHermanos (nivel, s)
                    Retroceder (nivel, s)
            finsi
    hasta nivel==0
```

- En algunos problemas los nodos intermedios pueden ser soluciones
- O bien, retroceder después de encontrar una solución

AND (nivel>0)

64

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Esquema general (*no recursivo*)

Caso 3) Problema de optimización (maximización).

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= sINICIAL
    voa:= -∞; soa:= Ø
    repetir
        Generar (nivel, s)
        si Solución (nivel, s) AND Valor(s) > voa entonces
            voa:= Valor(s); soa:= s
        si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT MasHermanos (nivel, s) AND (nivel>0)
                Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

voa: valor óptimo actual
soa: solución óptima actual

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Análisis general de tiempos de ejecución

- El tiempo de ejecución depende del número de nodos generados y del tiempo requerido para cada nodo, que viene dado por el coste de las funciones.
- Suponiendo que una solución sea de la forma: $(x_1, x_2, ..., x_n)$, en el peor caso se generarán todas las posibles combinaciones para cada $\mathbf{x_i}$.
- Si el número de posibles valores para cada $\mathbf{x_i}$ es $\mathbf{m_i}$, entonces se generan:

m_1	nodos en el nivel 1
$m_1 \cdot m_2$	nodos en el nivel 2
...	....
$m_1 \cdot m_2 \cdot \ldots \cdot m_n$	nodos en el nivel n

- **Ejemplo:** para el problema de la suma de subconjuntos $m_i = 2$. El número de nodos generados es:

$$t(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$$

- **Ejemplo**: calcular todas las permutaciones de (1, 2, ..., n). En el nivel 1 tenemos *n* posibilidades, en el nivel 2 *n-1*, ..., en el nivel *n* una posibilidad.

$$t(n) = n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \ldots + n! \in O(n!)$$

- **En general**, tendremos tiempos con órdenes de complejidad factoriales o exponenciales.

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio: Problema de la mochila

- **Datos del problema:**
  - **n:** número de TIPOS de objetos disponibles
  - **M:** capacidad de la mochila.
  - **u = (u_1,u_2, ..., u_n)** unidades de los objetos..
  - **b = (b_1,b_2, ..., b_n)** beneficios de los objetos.
- **Representación de la solución:**
  - Una solución será de la forma S = (x_1, x_2, ..., x_n), con $x_i \in$ **u**, siendo cada x_i unidades del objeto i.

Formulación matemática:

- Maximizar $\sum_{i=1}^{n} X_i b_i$

para i=1,2,…***n***

- Sujeto a la restricción $\sum_{i=1}^{n} X_i u_i \leq M \quad \text{para } i = 1..n$

Ejemplo:

Si **n** = 4,

M = 8,

u = (2, 3, 4, 5) unidades de los objetos,

b = (3, 5, 6, 10) beneficios de los objetos

67

Técnicas de diseño de algoritmos

Algoritmos Backtracking

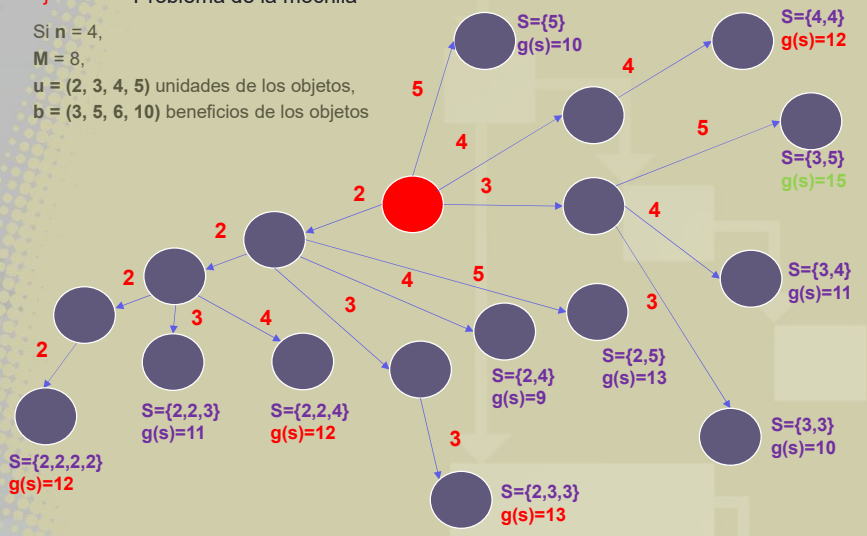
Ejercicio: Problema de la mochila

Si **n** = 4,

M = 8,

u = (2, 3, 4, 5) unidades de los objetos,

b = (3, 5, 6, 10) beneficios de los objetos



68

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Esquema general (*versión recursiva*)

Ejercicio: Problema de la mochila

función MochilaBackRec (u[], b[], Tipo *i*, Capacidad *r*)

```
{ Calcula el valor de la mejor carga que se puede construir
  empleando los elementos de tipos i a n y cuyo peso no sobrepase r }
a:=0
{ Se prueban por turno las clases de objetos admisibles }
para k:=i hasta n hacer
   si u[k] <= r entonces
      a:=max(a,  b[k] + MochilaBackRec(u, b, k, r-u[k])
devolver a
```

MochilaBackRec (1,M)

69

Técnicas de diseño de algoritmos

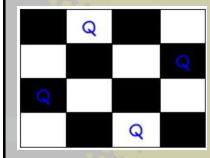
Algoritmos Backtracking

Ejercicio: Problema de las N-reinas

Las reinas en el ajedrez se pueden atacar horizontalmente, verticalmente y en diagonal.

El siguiente tablero muestra una solución al problema de ***N***-Reinas cuando n=8.

- El problema es cómo acomodar N reinas en el tablero sin que se ataquen entre ellas.
  - ¿Cómo realizar un algoritmo tipo backtracking que resuelva dicho problema?



Matriz de salida:
{0 , 1 , 0 , 0}
{0 , 0 , 0 , 1}
{1 , 0 , 0 , 0}
{0 , 0 , 1 , 0}

```
procedimiento reinas(k, col, diag45, diag135)
    // sol[1..k] es k-prometedor,
    // col = {sol[i] | 1 ≤ i ≤ k }
    // diag45 = {sol[i] –i+1| 1 ≤ i ≤ k } y diag135 = {sol[i] +i-1| 1 ≤ i ≤ k } }
    si k=8 entonces escribir sol
    si no {explorar las extensiones (k+1)-prometedoras de sol}
       para j←1 hasta 8 hacer
          si j ∉ col y j-k ∉ diag45 y j+k ∉ diag135
          entonces
             sol[k+1]←j
             reinas(k+1,col ∪ {j}, diag45 ∪ {j-k}, diag135 ∪ {j+k})
```


70

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio: Problema del recorrido de The Knight

La pieza se coloca en el primer bloque de un tablero vacío y, moviéndose de acuerdo con las reglas del ajedrez, debe visitar cada casilla exactamente una vez.

A continuación se muestra un tablero de ajedrez con 8 x 8 celdas. Los números en las celdas indican el número de movimiento de la pieza.

El algoritmo base es generar todos los recorridos uno por uno y comprobar si el recorrido generado cumple las restricciones.

```
mientras hay recorridos sin probar
{  generar el siguiente recorrido
   si este recorrido cubre todas las casillas
   {  imprime este recorrido;  }
}
```

Algoritmo:

si se visitan todas las casillas
    imprime la solución
sino

a) Agrega uno de los siguientes movimientos al vector solución y recursivamente verifica si este movimiento lleva a una solución (*Un Caballo puede hacer el máximo ocho movimientos. Elegimos uno de los 8 movimientos en este paso*).
b) Si el movimiento elegido en el paso anterior no lleva a una solución, entonces elimina este movimiento del vector solución y prueba con otros movimientos alternativos.
c) Si ninguna de las alternativas funciona, regresa "falso" (*Devolver falso eliminará el elemento previamente agregado en recursividad y si es falso se devuelve por la llamada inicial de recursividad, entonces "no existe ninguna solución"*)

Backtracking Algorithms Explained

71

Técnicas de diseño de algoritmos

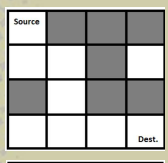
Algoritmos Backtracking

Ejercicio: Problema de la rata en un laberinto

Un laberinto se da como una matriz binaria de bloques N * N donde el bloque de origen es el bloque superior a la izquierda, es decir, el laberinto [0] [0] y el bloque de destino es el bloque inferior a la derecha, es decir, laberinto [N-1] [N-1].

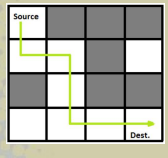
Una rata parte de la fuente y tiene que llegar al destino. La rata solo puede moverse en dos **direcciones**: hacia adelante y hacia abajo.

En la matriz del laberinto, 0 significa que el bloque es un callejón sin salida y 1 significa que el bloque se puede utilizar en la ruta desde el origen hasta el destino.



Matriz de entrada

{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}



Matriz de salida

{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}

Algoritmo:

1. Declara una matriz de solución, inicia con ceros.
2. Define una función recursiva, que toma la matriz inicial, la matriz de salida y la posición de rat (i, j).
3. Si la posición está fuera de la matriz o la posición no es válida, regresa.
4. Marca la salida de posición [i] [j] como 1 y comprueba si la posición actual es el destino o no. Si es el destino, muestra la matriz de salida.
5. Recursivamente llama la posición (i + 1, j) y a (i, j + 1).
6. Desmarcar la posición (i, j), es decir, salida [i] [j] = 0.

Una versión más compleja puede ser que la rata se pueda mover en 4 direcciones y una versión más compleja puede ser con un número limitado de movimientos.

Introduction to Backtracking | Rat in a Maze | Backtracking-2

72

Técnicas de diseño de algoritmos

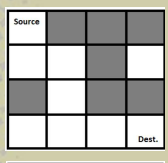
Algoritmos Backtracking

Ejercicio: Problema de la rata en un laberinto

Un laberinto se da como una matriz binaria de bloques N * N donde el bloque de origen es el bloque superior a la izquierda, es decir, el laberinto [0] [0] y el bloque de destino es el bloque inferior a la derecha, es decir, laberinto [N-1] [N-1].

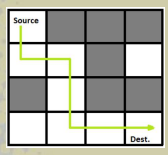
Una rata parte de la fuente y tiene que llegar al destino. La rata solo puede moverse en dos direcciones: hacia adelante y hacia abajo.

En la matriz del laberinto, 0 significa que el bloque es un callejón sin salida y 1 significa que el bloque se puede utilizar en la ruta desde el origen hasta el destino.



Matriz de entrada

{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}



Matriz de salida

{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}

Algoritmo:

```
inicio
    si (x, y) es la esquina inferior derecha, entonces
        marca el lugar como 1
        regresa true
    si esLugarValido(x, y) = true, entonces
        marca(x, y) lugar como 1
        si solLaberinto(x + 1, y) = true, entonces  // mov. hacia adel
            regresa true
        si solLaberinto(x, y + 1) = true, entonces // mov. hacia abajo
            regresa true
        marca(x, y) como 0 cuando se retrocede
        regresa falso
    regresa falso
fin
```

Introduction to Backtracking Rat in a Maze | Backtracking-2 Rat in a Maze Problem

73

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Algunas aplicaciones:

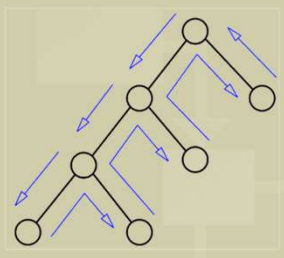
- Ciclo hamiltoniano
- Problema de colorear M
- Rompecabezas criptográfico
- Problema de suma de subconjuntos
- Algoritmo de resolución de sudoku
- Problema de tira y afloja
- Algoritmo de ruptura de palabras (*word break*)
- Número máximo para problemas de intercambio

Conclusiones

- El retroceso es una técnica algorítmica para resolver problemas de forma recursiva al intentar construir una solución de forma incremental.
- Resolver una pieza a la vez y eliminar aquellas soluciones que no satisfacen las limitaciones del problema en cualquier momento (por tiempo, aquí, se refiere al tiempo transcurrido hasta alcanzar cualquier nivel del árbol de búsqueda) es el proceso de retroceso.

Introduction to Backtracking Rat in a Maze | Backtracking-2

74



Ramificación y acotamiento(poda)
(Branch and bound)

75

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) *(Branch and Bound)*

- *Branch and bound* es una técnica general para mejorar el proceso de búsqueda mediante la enumeración sistemática de todas las soluciones candidatas y la eliminación de soluciones obviamente imposibles.
  - Es una extensión de *backtracking*, en el cual se recorre el árbol de soluciones como en *retroceso*, pero se recuerda el mejor valor encontrado hasta el momento.
- Generalmente se aplica a aquellos problemas que tienen soluciones finitas, en las que las soluciones se pueden representar como una secuencia de opciones, como los problemas de optimización combinatoria.
  - Los problemas de optimización, suelen ser exponenciales en términos de complejidad de tiempo y pueden requerir explorar todas las posibles permutaciones en el peor de los casos; esta técnica resuelve estos problemas con relativa rapidez.
- El espacio de la solución se organiza como una estructura en forma de árbol.
- **Ejemplo:** el problema del vendedor viajero (ambulante) que intenta encontrar el recorrido más corto por las ciudades. *Avanzar por una rama determinada equivale a decidir en qué orden se visitan las ciudades.*

Branch and Bound Algorithm

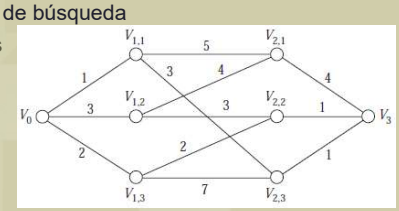
76

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) *(Branch and Bound)*

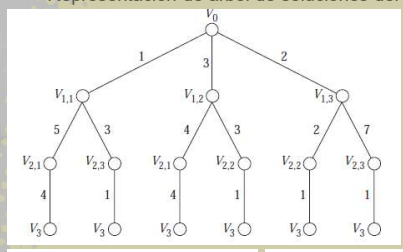
Ejercicio: Problema multietapas de una gráfica de búsqueda

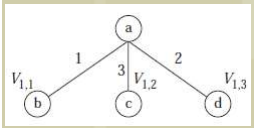
El problema consiste en encontrar una ruta más corta de V_0 a V_3.

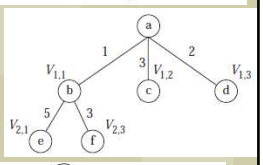


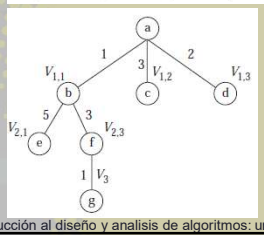
Solución:

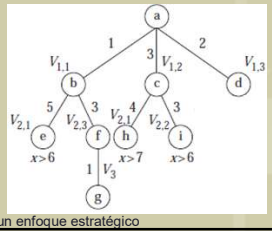
Representación de árbol de soluciones del problema

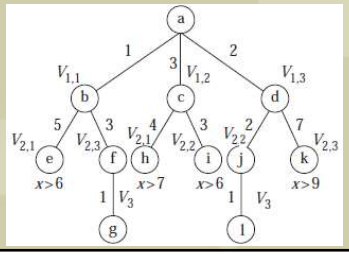












Introducción al diseño y analisis de algoritmos: un enfoque estratégico

77

Técnicas de diseño de algoritmos

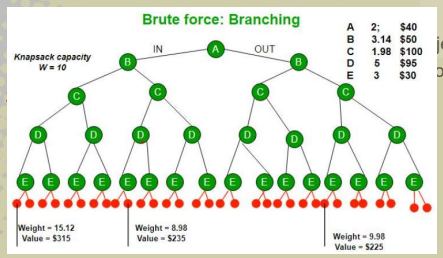
Algoritmos Ramificación y acotar (poda) *(Branch and Bound)*

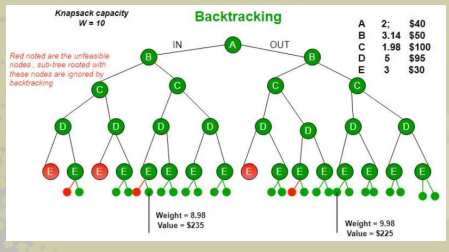
Ejercicio: Problema de la mochila

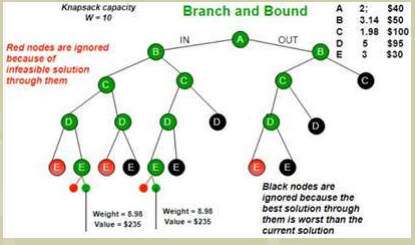
Datos del problema:

- **n = 5;** número de TIPOS de objetos disponibles

A	2;	$40
B	3.14	$50
C	1.98	$100
D	5	$95
E	3	$30







Branch and Bound Algorithm

78

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) *(Branch and Bound)*

Ejercicio: Problema de la mochila

Datos del problema:

- **n = 5;** número de TIPOS de objetos disponibles
- **M = 10;** capacidad de la mochila
- **u = (2, 3.14, 1.98, 5, 3)** unidades de los objetos
- **b = (40, 50, 100, 95, 30)** beneficios de los objetos

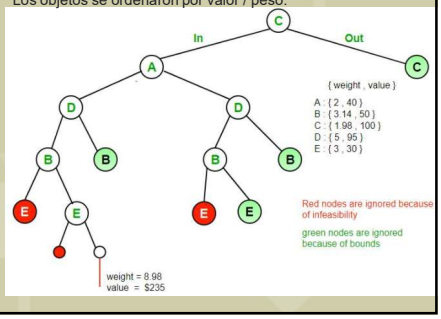
A	2;	$40
B	3.14	$50
C	1.98	$100
D	5	$95
E	3	$30

Solución:

1. Ordena todos los elementos en orden decreciente relacionando el valor por unidad de peso para que se pueda calcular un límite superior utilizando el enfoque codicioso.
2. Inicializa el beneficio máximo, maxProfit = 0
3. Crea una cola vacía, Q.
4. *Crea un nodo ficticio de árbol de decisión y colócalo en Q. La ganancia y el peso del nodo ficticio son 0.*
5. Mientras Q no esté vacío, se realizará:
   - Extraer un elemento de Q, le llamaremos *u*.
   - Calcula el beneficio del nodo del siguiente nivel. *Si el beneficio es superior a maxProfit, actualiza maxProfit.*
   - Calcula el límite del nodo del siguiente nivel. *Si el límite es mayor que maxProfit, agrega el siguiente nodo de nivel a Q.*
   - Considera que el nodo del siguiente nivel no se considera parte de la solución y agrega un nodo a la cola con el nivel siguiente, pero el peso y la ganancia no considera los nodos del siguiente nivel.

Entrada:
// Para cada par se obtiene el peso de los objetos
// y después el valor del mismo
Elemento arr[ ] = {{2, 40}, {3.14, 50}, {1.98, 100}, {5, 95}, {3, 30}};
Capacidad de la mochila W = 10
Salida: El beneficio máximo posible = 235

Los objetos se ordenaron por valor / peso.



Implementation of 0/1 Knapsack using Branch and Bound

79

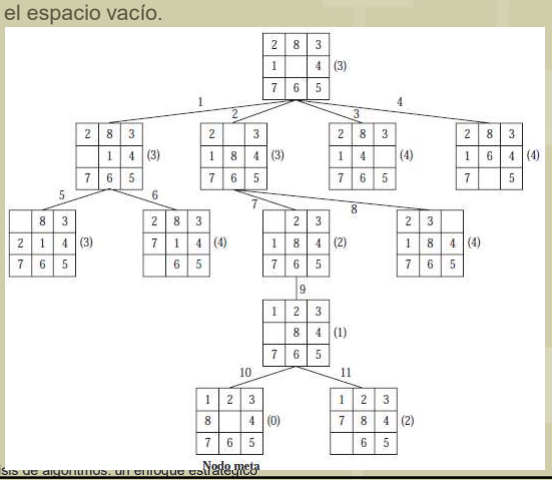
Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) *(Branch and Bound)*

Ejercicio: Problema de rompecabezas de 8 piezas

Dado un tablero de 3 × 3 con 8 fichas (*cada ficha tiene un número del 1 al 8*) y un espacio vacío. El objetivo es colocar los números en los mosaicos para que coincidan con la configuración final utilizando el espacio vacío.
Se pueden deslizar cuatro fichas adyacentes (*izquierda, derecha, arriba y abajo*) en el espacio vacío.

Solución:



Nodo meta

Introducción al diseño y análisis de algoritmos: un enfoque estratégico

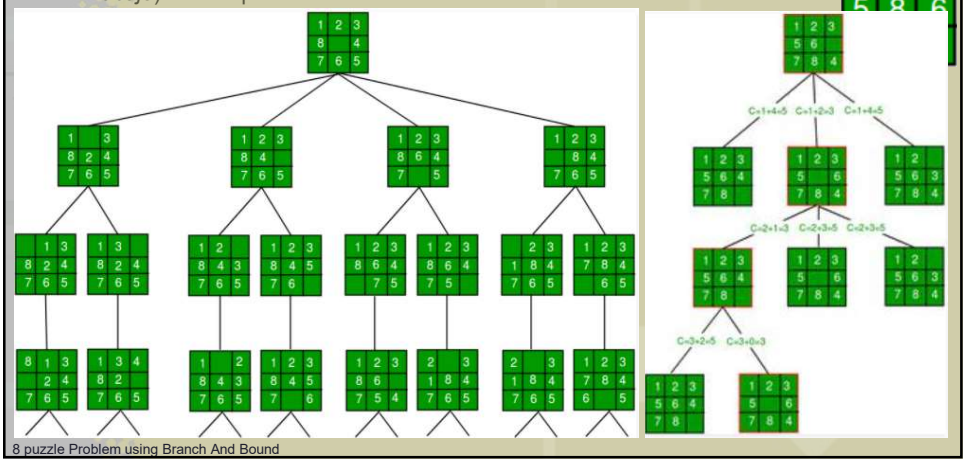
80

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) (*Branch and Bound*)

Ejercicio: Problema de rompecabezas de 8 piezas

Dado un tablero de 3 × 3 con 8 fichas (*cada ficha tiene un número del 1 al 8*) y un espacio vacío. El objetivo es colocar los números en los mosaicos para que coincidan con la configuración final utilizando el espacio vacío.
Se pueden deslizar cuatro fichas adyacentes (*izquierda, derecha, arriba y abajo*) en el espacio vacío.



8 puzzle Problem using Branch And Bound

81

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) (*Branch and Bound*)

Algunas aplicaciones:

- Problema de la mochila
- Rompecabezas
- Asignación de trabajos
- Problema de las N-reinas
- Problema del vendedor viajero

Conclusiones

- Branch-and-bound es una técnica general para mejorar el proceso de búsqueda mediante la enumeración sistemática de todas las soluciones candidatas y la eliminación de soluciones obviamente imposibles.
- Se aplica a aquellos problemas que tienen soluciones finitas, en las que las soluciones se pueden representar como una secuencia de opciones.
- La primera parte del proceso requiere que se realicen varias elecciones para que se ramifiquen en el espacio de la solución.
- En estos métodos, el espacio de la solución se organiza como una estructura en forma de árbol. La diversificación de todas las opciones posibles garantiza que no se dejará al descubierto ninguna solución potencial.
- Debido a que el problema objetivo suele ser NP-completo, el espacio de la solución suele ser demasiado amplio para atravesarlo.
- Una ventaja importante de estos algoritmos es que se puede controlar la calidad de la solución que se espera, incluso si aún no se ha encontrado.

Branch and Bound Algorithm Branch and Bound Algorithm

82

Técnicas de diseño de algoritmos

Algoritmos Backtracking *vs* Ramificación y acotar (poda)

Backtracking:

- *Backtracking* es un algoritmo general para encontrar todas las soluciones a algunos problemas computacionales, en particular:
  - problemas de **satisfacción de restricciones**, que construye gradualmente posibles candidatos a las soluciones y abandona a un candidato tan pronto como determina que el candidato no puede ser completado para finalmente convertirse en un solución válida.
- Es una técnica algorítmica para resolver problemas de forma recursiva al tratar de construir una solución de forma incremental, una pieza a la vez, eliminando aquellas soluciones que no logran satisfacer las restricciones del problema en cualquier momento.

Ramificación y acotación (poda)

- *Branch and bound* es un paradigma de diseño de algoritmos para problemas de **optimización** discretos y combinatorios, así como optimización matemática.
- El algoritmo consiste en una enumeración sistemática de soluciones candidatas:
  - Es decir, se piensa que el conjunto de soluciones candidatas forma un árbol enraizado con el conjunto completo en la raíz.
  - Se exploran las ramas de este árbol, que representan los subconjuntos del conjunto de soluciones.
  - Antes de enumerar las soluciones candidatas de una rama, la rama se compara con los límites estimados superior e inferior de la solución óptima y se descarta si no puede producir una solución mejor que la mejor encontrada hasta ahora por el algoritmo.

Difference between Backtracking and Branch-N-Bound technique

83

Técnicas de diseño de algoritmos

Algoritmos Backtracking *vs* Ramificación y acotar (poda)

Característica	Backtracking	Branch and Bound
Approach	• Se utiliza para encontrar todas las posibles soluciones disponibles a un problema. Cuando se da cuenta de que ha hecho una mala elección, deshace la última elección haciendo una copia de seguridad. • Busca en el árbol del espacio de estados hasta que encuentra una solución al problema.	• Se utiliza para resolver problemas de optimización. • Cuando se da cuenta de que ya tiene una mejor solución óptima a la que conduce la pre-solución, abandona esa pre-solución. • Busca por completo el árbol espacial de estados para obtener una solución óptima.
Problemas	Resuelve problemas de decisión.	Resuelve problemas de optimización.
Función	Función de viabilidad.	Función de límite.
Búsqueda	Se busca en el árbol del espacio de estados hasta que se obtiene la solución.	La solución óptima puede estar presente en cualquier parte del árbol del espacio de estado, por lo que es necesario buscar en el árbol por completo.
Recorrido	Recorre el árbol del espacio de estado de manera profunda.	Recorre el árbol de cualquier manera, a profundidad y limitando el árbol.
Aplicaciones	N-Queen, suma del subconjunto.	Problemas de mochila, problema de vendedor ambulante.

Difference between Backtracking and Branch-N-Bound technique

84