

1

# Técnicas de Concepção de Algoritmos (1ª parte): algoritmos de retrocesso

R. Rossetti, A.P. Rocha, J. Pascoal Faria  
CAL, MIEIC, FEUP  
Fevereiro de 2014

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

2

## Para pensar...

- ◆ “Theory is when you know something, but it doesn’t work.  
Practice is when something works, but you don’t know why.  
Programmers combine theory and practice: Nothing works and they don’t know why.”

(unknown)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

3

## Algoritmos de retrocesso (*backtracking*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

4

## Algoritmos de retrocesso

- ◆ Um dado **problema** tem um conjunto de restrições e possivelmente uma função objectivo
- ◆ Uma **solução** otimiza a função objectivo e/ou a satisfaz
- ◆ Pode-se representar o espaço de solução para o problema utilizando-se uma árvore de espaço de estados
  - A raiz da árvore representa 0 escolhas
  - Nós ao nível 1 representam primeira escolha
  - Nós ao nível 2 representam segunda escolha, etc...
- ◆ O caminho da raiz a uma folha representa uma solução candidata

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

5

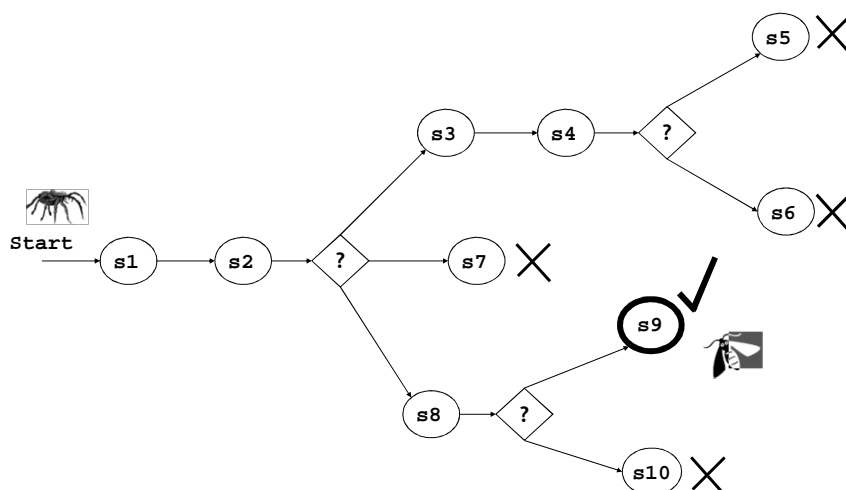
## Algoritmos de retrocesso

- ◆ Algoritmos de *tentativa e erro*
- ◆ Contexto geral de aplicação:
  - Explorar um *espaço de estados* à procura dum *estado-objectivo*
  - Estado = estado de jogo, sub-problema a resolver, posição, etc.
  - Sem algoritmos eficientes que levem directamente ao objectivo
- ◆ Estratégia:
  - Ao chegar a um *ponto de escolha* (c/ vários estados seguintes), escolher uma das opções e prosseguir a exploração
  - Chegando a um “beco sem saída”, *retroceder* até ao ponto de escolha + próximo c/alternativas p/explorar, e tentar outra alt.
- ◆ Exemplos:
  - Problema do troco quando há falta de stock de algumas moedas
  - Sudoku, 8 rainhas, *puzzles* em geral.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

6

## Ilustração



Complicações: ciclos, caminhos paralelos

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

7

## Implementação recursiva

- ◆ Implementado normalmente de forma recursiva
  - avanço corresponde a uma chamada recursiva
  - retrocesso corresponde ao retorno de chamadas recursivas

Explore state/node N:

1. if N is a goal state/node, return “success”
2. (optional) if N is a leaf state/node, return “failure”
3. for each successor/child C of N,
  - 3.1. (if appropriate) set new state
  - 3.2. explore state/node C
  - 3.3. if exploration was successful, return “success”
  - 3.4 (if step 3.1 was performed) restore previous state
4. return “failure”

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

8

## Ex. Soma de subconjuntos

- ◆ Problema: dados  $n$  positivos inteiros  $w_1, \dots, w_n$  e um inteiro positivo  $S$ , encontrar todos os subconjuntos de  $w_1, \dots, w_n$  cuja soma é  $S$
- ◆ Exemplo:  $n = 3, S = 6, W = \{2, 4, 6\}$
- ◆ Solução:
  - $\{2, 4\}$
  - $\{6\}$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

9

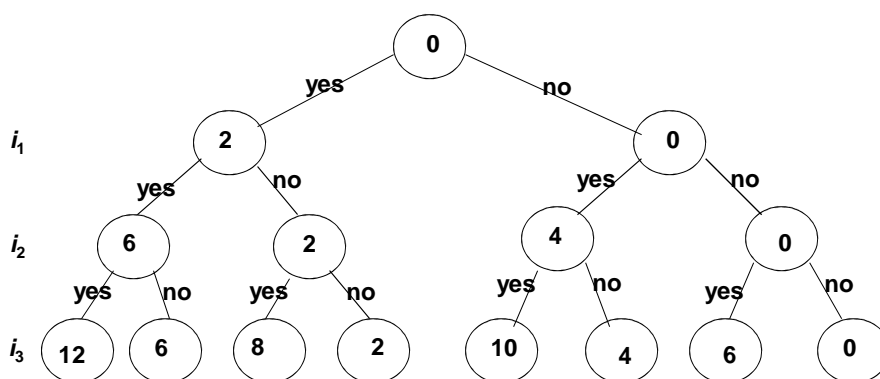
## Ex. Soma de subconjuntos

- ◆ Para este caso, assume-se uma árvore binária para o espaço de estados
- ◆ Nós ao nível 1 representam incluir (sim ou não) o item 1, nós ao nível 2 representam incluir item 2, etc...
- ◆ O ramo esquerdo da árvore inclui  $w_1$  enquanto o ramo direito da árvore exclui  $w_1$
- ◆ Os nós contêm as somas dos pesos incluídos até então!

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

10

**Problema: soma de subconjuntos**  
**Árvore de espaço de estados para 3 itens**  
 $w_1 = 2, w_2 = 4, w_3 = 6$  e  $S = 6$



**A soma dos inteiros incluídos é guardada nos nós!**

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

11

## Ex. Soma de subconjuntos

- ◆ Problemas como este podem ser resolvidos realizando-se uma pesquisa/busca em profundidade
- ◆ Cada nó guardará o seu nível (profundidade) e a sua solução (possivelmente parcial) corrente
- ◆ Uma busca em profundidade pode verificar se um nó  $v$  é uma folha:
  - Se  $v$  é uma folha, então verifica-se se a solução corrente satisfaz as restrições do problema
  - Extensões a este método podem ser implementadas a fim de se encontrar uma solução ótima

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

12

## Ex. Soma de subconjuntos

- ◆ Uma estratégia baseada unicamente em busca/pesquisa em profundidade pode representar uma alternativa muito cara em termos de tempo de processamento!
- ◆ Neste caso, não se verifica para todo estado solução (nó) quando a solução foi alcançada, ou mesmo se uma solução parcial poderá levar a uma solução satisfatória
- ◆ Questão: é possível desenvolver uma alternativa mais eficiente?

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

13

## Estratégia de Retrocesso

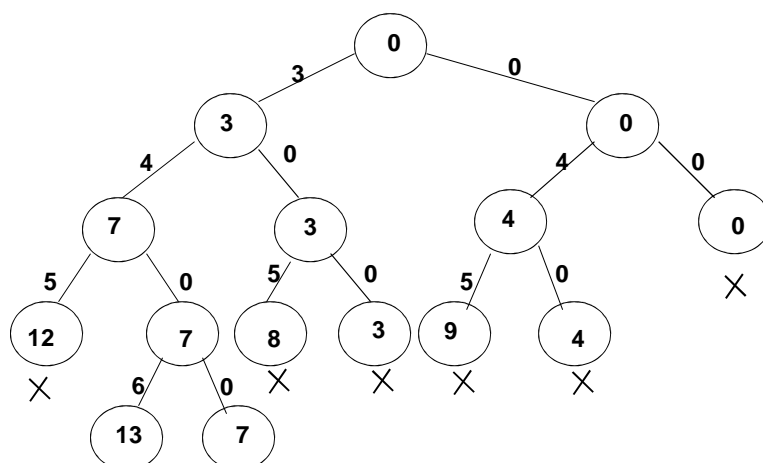
- ◆ Definição: chama-se a um nó “não promissor” caso este não conduza a uma solução viável (ou óptima). Caso contrário, este será tido como um nó “promissor”
- ◆ Ideia básica: retrocesso consiste em realizar uma pesquisa em profundidade na árvore de espaço de estados, verificando se um nó é promissor, e caso o nó não seja promissor, retroceder até o nó pai.
- ◆ A uma árvore de espaço de estados que contém apenas nós expandidos chama-se árvore de espaço de estados podada

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

14

Árvore podada de espaço de estados (p/ encontrar todas as soluções)

$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

15

## Estratégia de Retrocesso

```
void checknode (node v) {  
    node u  
  
    if ( promising ( v ) )  
        if ( aSolutionAt( v ) )  
            write the solution  
        else //expand the node  
            for ( each child u of v )  
                checknode ( u )  
}
```

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

16

## Estratégia de Retrocesso

◆ *Checknode* usa as funções:

- *promissing*( *v* ) que verifica se a solução parcial representada pelo nó *v* poderá levar à solução desejada
- *aSolutionAt*( *v* ) que verifica se a solução parcial representada pelo nó *v* resolve o problema em questão

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014



17

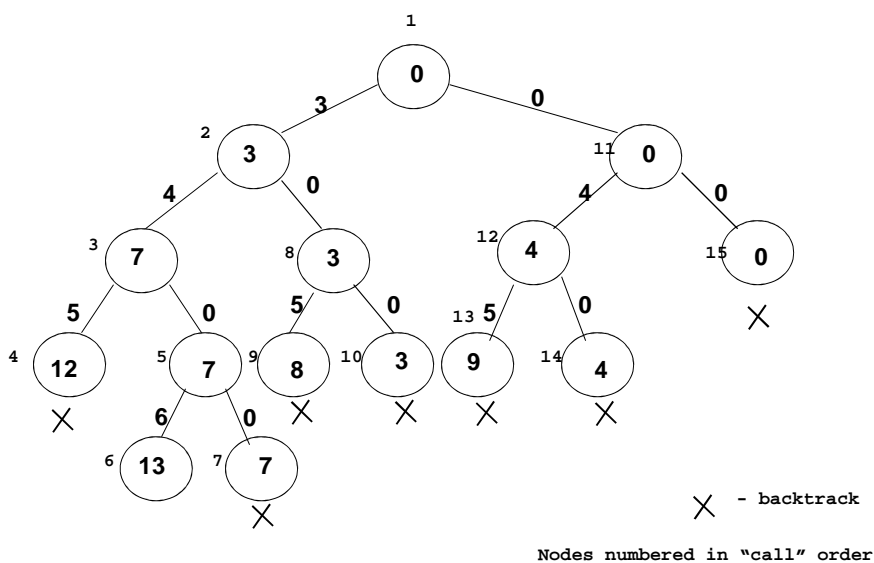
## Estratégia de Retrocesso

- ◆ Quando um nó é “promissor”?  
Considere um nó ao nível  $i$ :
  - *weightSoFar*: peso do nó, i.e. soma dos números incluídos na solução parcial que o nó representa
  - *totalPossibleLeft*: peso dos itens remanescentes ( $i + 1$  a  $n$ ) para um nó ao nível  $i$
  - Um nó ao nível  $i$  é “não promissor” se  $\text{weightSoFar} + \text{totalPossibleLeft} < S$  (ou)  $\text{weightSoFar} + w[i + 1] > S$
  - Para se poder utilizar a função *promissing*, os elementos  $w_i$  devem estar ordenados numa ordem não decrescente!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

18

Árvore podada de espaço de estados  
 $w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

19

```

sumOfSubsets ( i, weightSoFar, totalPossibleLeft )
  1) if (promising ( i ))                //may lead to solution
  2) then if ( weightSoFar == S )
  3)   then print include[ 1 ] to include[ i ] //found solution
  4) else //expand the node when weightSoFar < S
  5)   include [ i + 1 ] = "yes"          //try including
  6)   sumOfSubsets ( i + 1, weightSoFar + w[i + 1],
                        totalPossibleLeft - w[i + 1] )
  7)   include [ i + 1 ] = "no"           //try excluding
  8)   sumOfSubsets ( i + 1, weightSoFar ,
                        totalPossibleLeft - w[i + 1] )

```

boolean promising ( i )

```

  1) return ( weightSoFar + totalPossibleLeft ≥ S ) &&
        ( weightSoFar == S || weightSoFar + w[i + 1] ≤ S )

```

Prints all solutions!

Chamada inicial da função sumOfSubsets(0, 0,  $\sum w_i$ )

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

20

## Problemas de optimização

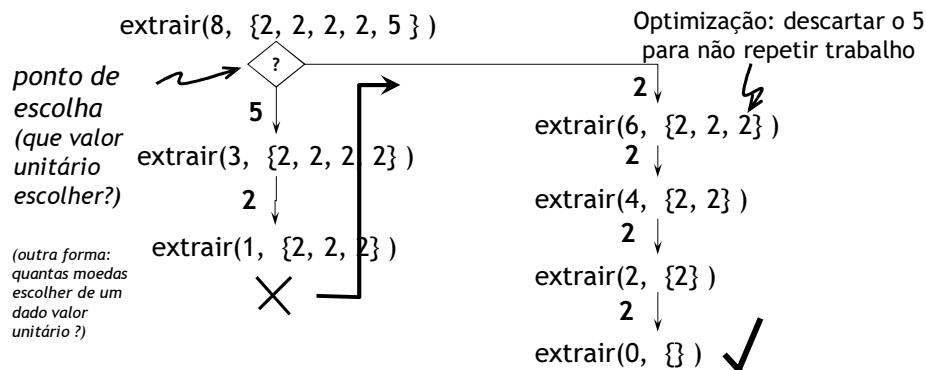
- ◆ Para problemas de optimização, considera-se também:
  - *best* - valor da melhor solução encontrada até então
  - *value( v )* - valor da solução no nó *v*
  - Deve-se modificar a função *promissing( v )*
  - *best* é inicializado com um valor igual a uma solução candidata ou pior que qualquer uma solução possível
  - *best* é actualizado com *value( v )* se a solução em *v* é “melhor”
  - Ser “melhor” dependerá do problema (maximização ou minimização)!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

21

## Exemplo: Problema do troco

- ◆ Na falta de stock de algumas moedas, o algoritmo ganancioso pode não dar solução, quando ela existe (\*)
- ◆ Resolução: retroceder até ao ponto de escolha mais próximo e escolher a moeda de valor mais baixo a seguir



(\*) também pode dar uma solução não ótima, mas isso resolve-se de outra forma - com programação dinâmica

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

22

## Implementação recursiva

```
static final int moedas[] = {1,2,5,10,20,50,100,200};

// stock[i] = nº de moedas de valor moedas[i]
public int[] select(int montante, int[] stock) {
    int[] sel = new int[moedas.length];
    return select(montante, stock, sel, moedas.length-1)? sel:null;
}

boolean select(int mont, int[] stock, int[] sel, int maxIdx) {
    /*1.*/ if (mont == 0)
        return true;
    /*3.*/ for (int i = maxIdx; i >= 0; i--)
        if (stock[i] > sel[i] && moedas[i] <= mont) {
            /*3.1.*/ sel[i]++; mont -= moedas[i];
            /*3.2.*/ if (select(mont, stock, sel, maxIdx))
                return true;
            /*3.3.*/ sel[i]--; mont += moedas[i];
        }
    /*4.*/ return false;
}
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

24

## Poda da pesquisa

- ◆ Interromper a pesquisa em ramos que garantidamente não levam a nenhuma solução
- ◆ Exemplo no problema do troco: interromper a pesquisa (e retornar indicação de insucesso) quando o valor do stock utilizável é inferior ao montante em falta

```
boolean select(int mont, int[] stock, int[] sel, int maxIdx) {
    if (mont == 0)
        return true;
    if (mont > total(stock,0,maxIdx)-sel[maxIdx]*moedas[maxIdx])
        return false;
    ...
}
```
- ◆ Melhora o desempenho mas podem continuar a existir casos patológicos com tempo de execução exponencial

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

25

## Variantes de aplicação

- ◆ Encontrar uma solução (caso estudado até aqui)
  - A pesquisa pára assim que se encontra a primeira solução
- ◆ Encontrar todas as soluções
  - Quando se encontra uma solução, processa-se essa solução (imprimir, etc.), mas não se pára a exploração
  - Retrocede-se para o ponto de escolha mais próximo como se tivéssemos chegado a um “beco sem saída”
- ◆ Encontrar a melhor solução
  - Variante de encontrar todas as soluções, em que se vai guardando a melhor solução encontrada até ao momento
  - Podar a pesquisa: interromper um caminho de pesquisa quando temos a certeza que não permite chegar a uma solução melhor

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2014

## Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992

