1

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

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão
CAL, MIEIC, FEUP

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)

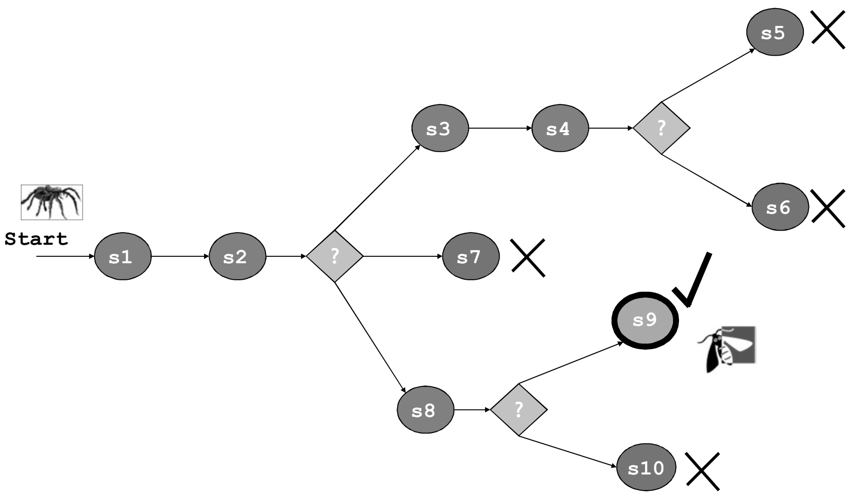
Algoritmos de retrocesso (*backtracking*)

Algoritmos de retrocesso

- Algoritmos de *tentativa e erro*
- Contexto geral de aplicação:
  - Explorar um *espaço de estados* à procura dum *estado-objetivo*
  - Estado = estado de jogo, subproblema, solução parcial, etc.
  - Sem algoritmos eficientes que levem directamente ao objetivo
- Estratégia:
  - Ao chegar a um *ponto de escolha* (com vários estados seguintes), escolher uma das opções e prosseguir a exploração
  - Chegando a um "beco sem saída", *retroceder* até ao ponto de escolha mais próximo com alternativas por explorar, e tentar outra alternativa

5

Ilustração



Complicações: ciclos, caminhos paralelos

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

6

Exemplos de aplicação

- Problema do troco com limitações de stock
- Sudoku
- 8 rainhas
- Labirintos

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

7

Geração do espaço de estados

- Dado um **problema** com um conjunto de restrições e/ou uma função objetivo (maximizar ou minimizar).
- Procura-se uma **solução** que satisfaz as restrições e/ou optimize a função objetivo.
- Caso a solução possa ser construída passo a passo …
- Pode-se representar (pelo menos concetualmente) o espaço de solução para o problema através de uma árvore de espaço de estados
  - A raiz da árvore representa o início (0 escolhas)
  - Nós ao nível 1 representam a primeira escolha
  - Nós ao nível 2 representam a segunda escolha, etc…
  - Caminhos da raiz até às folhas representam soluções candidatas

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

8

Exemplo: soma de subconjuntos

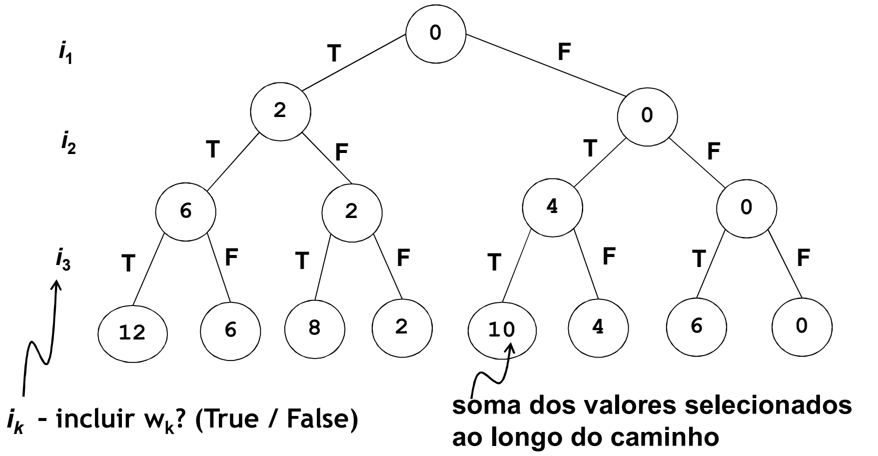
- Problema: dado um conjunto (ou multiconjunto) $W=\{w_1, ..., w_n\}$ de inteiros positivos e uma soma S a perfazer, encontrar um subconjunto R de W com soma S
- Exemplo: **W** = {2, 4, 6} e S = 6
- Solução: {2, 4} ou {6}

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

9

Árvore de espaço de estados binária

- Uma possibilidade é uma árvore binária em que, em cada nível k, se decide da inclusão ou não do valor w_k
- As folhas representam as soluções candidatas.



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

10

Algoritmo geral de pesquisa

- Algoritmo recursivo de visita em profundidade da árvore de espaço de estados
  - avanço corresponde a uma chamada recursiva
  - retrocesso corresponde a retorno de chamada recursiva falhada

```
Explore state/node N:
1.  if N is a goal state/node, return "success"
2.  for each successor/child C of N,
    2.1.  (optional) set new state
    2.2.  explore state/node C
    2.3.  if exploration was successful, return "success"
    2.4   (optional) restore previous state
3.  return "failure"
```

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

11

Exemplo de implementação

```
//Inputs: int W[], int n, int S
//Oututs: bool sel[] - sel[i]=true means W[i] selected (initially false)
//Call initially: sumOfSubsets(0, 0)
bool sumOfSubsets(int i, int sumSel) {
   // if solution found, print and terminate
   if (sumSel == S) {printSolution(); return true;}

   // if there is no child to explore, just backtrack
   if (i == n)  return false;

   // explore item W[i] (try using and not using) (choice point)
   sel[i] = true;
   if (sumOfSubsets(i+1, sumSel + W[i])) return true;
   sel[i] = false;
   if (sumOfSubsets(i+1, sumSel)) return true;

   // no solution found
   return false;
}
```

Sum of items already selected

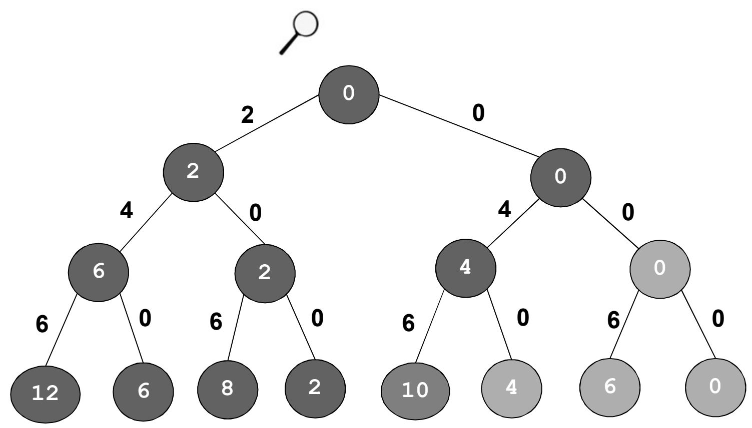
Index of next item to explore (=level in search tree).

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

12

Exemplo de exploração

W = {2, 4, 6}; S = 10



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

Eficiência temporal

- Tempo de execução no pior caso (pesquisa exaustiva do espaço de estados) é determinado pela dimensão do espaço de estados, que muitas vezes é exponencial
- Exemplo no problema da soma de subconjuntos:
  - As folhas da árvore binária representam os possíveis subconjuntos de *W*, em número $\# \mathcal{P}(W) = 2^{\#W} = 2^{n}$
  - O nº de nós da árvore é sensivelmente o dobro (2^{n+1}-1)
  - Se a soma *S* a perfazer for próxima de *sum(W)*, mas não for realizável, resulta uma pesquisa exaustiva ou quase!
  - No pior caso, tempo portanto T(n) = O(2^n)
- Como se pode melhorar?

Poda da pesquisa (*pruning*)

- Interromper (podar) a pesquisa e retroceder em nós que garantidamente não levam a uma solução viável (chamados nós não promissores)
- No problema da soma de subconjuntos, pode-se podar a pesquisa quando:
  - a soma já selecionada é superior à soma a perfazer
  - a soma ainda selecionável é inferior à soma a perfazer
- A uma árvore de espaço de estados que contém apenas nós expandidos chama-se árvore de espaço de estados podada

Algoritmo geral

```
exploreNode(v):
    if aSolutionAt(v) then
        write the solution and terminate
    else if promising(v) then
        for each u ∈ child(v) do
            exploreNode(u)
```

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

Exemplo de implementação

```
// Call initially: sumOfSubsets(0, 0, sum(w,n))
bool sumOfSubsets(int i, int sumSel, int sumLeft) {
   // if solution found, print and terminate
   if (sumSel == S) {printSolution(); return true;}

   // if there is no child to explore, just backtrack
   if (i == n)  return false;

   // if not a promising state, prune the search
   if (sumSel + sumLeft < S || sumSel + w[i] > S) return false;

   // explore item W[i] (try using and not using) (choice point)
   sel[i] = true;
   if (sumOfSubsets(i+1, sumSel+W[i], sumLeft-W[i])) return true;
   sel[i] = false;
   if (sumOfSubsets(i+1, sumSel, sumLeft-W[i])) return true;

   // no solution found
   return false;
}
```

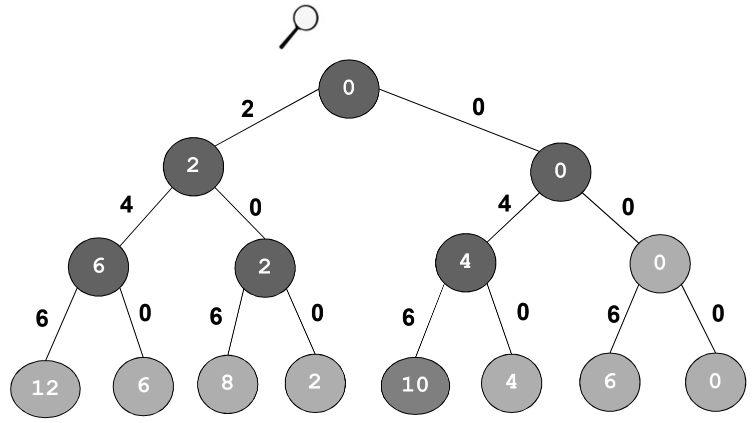
Soma dos itens remanescentes

Assumindo itens por ordem crescente

17

Exemplo de poda

W = {2, 4, 6}; S = 10



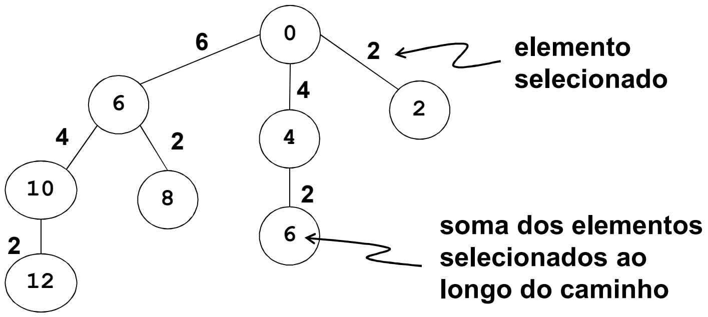
18

Variantes do problema

- Encontrar uma solução (caso considerado 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, guardar, etc.), mas não se pára a exploração
- Encontrar a melhor solução
  - Exemplo: encontrar um subconjunto de *W* de cardinal mínimo e soma *S* (como no problema do troco)
  - Variante de encontrar todas as soluções, em que se vai guardando a melhor solução encontrada até ao momento
  - Pode-se podar a pesquisa quando um nó não leva a uma solução melhor que a(s) encontrada(s) até ao momento

Outras árvores de espaço de estados

- Outra possibilidade no problema da soma de subjconjuntos: árvore n-ária em que, em cada nível, se escolhe o próximo valor a incluir.
- Para evitar repetir soluções, valores são escolhidas por ordem crescente ou decrescente.
- Cada nó representa uma solução candidata (total 2^n).



Outras otimizações

- Combinar com algoritmo ganancioso para procurar chegar mais rapidamente à solução (ou a uma boa solução quando se procura o ótimo)
  - Por exemplo, no problema do troco ou da soma de subconjuntos, começar a pesquisa pelos valores mais elevados
- Combinar com técnicas de memorização para evitar explorar repetidamente o mesmo nó, na presença de caminhos paralelos ou ciclos
  - Por exemplo, ao pesquisar num espaço de estados em forma de grafo, marcar os nós já visitados
- Em combinação com técnica de poda da pesquisa, podem melhorar o desempenho mas podem continuar a existir casos patológicos com tempo exponencial

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