

IFT2125 - Introduction à l'algorithmique

Exploration de graphes (B&B chapitre 9)

Pierre McKenzie

DIRO, Université de Montréal

Hiver 2018

Fouille en profondeur d'un graphe

B&B sections 9.3 et 9.4

```
procedure dfsearch(G)  
  for each  $v \in N$  do  $mark[v] \leftarrow not\text{-}visited$   
  for each  $v \in N$  do  
    if  $mark[v] \neq visited$  then dfs( $v$ )
```

```
procedure dfs( $v$ )  
  {Node  $v$  has not previously been visited}  
   $mark[v] \leftarrow visited$   
  for each node  $w$  adjacent to  $v$  do  
    if  $mark[w] \neq visited$  then dfs( $w$ )
```

Fouille en profondeur

Quelques utilités

- Dans un graphe non-orienté, l'arbre de la fep peut servir à
 - ▶ calculer les composantes connexes,
ensemble maximal de sommets reliés deux à deux par un chemin
 - ▶ calculer les points d'articulation.
sommets qui, retirés, brisent la connexité
- Dans un graphe orienté, l'arbre de la fep peut servir à
 - ▶ détecter un cycle,
 - ▶ sinon à trier les sommets en ordre "topologique".
l'arc $s \rightarrow s'$ implique $s \leq s'$
- Pire cas et meilleur cas $\Theta(\max(\# \text{ d'arcs}, \# \text{ de sommets}))$.

Fouille en largeur d'un graphe

B&B section 9.5

```
procedure bfs( $v$ )  
   $Q \leftarrow \text{empty-queue}$   
   $\text{mark}[v] \leftarrow \text{visited}$   
  enqueue  $v$  into  $Q$   
  while  $Q$  is not empty do  
     $u \leftarrow \text{first}(Q)$   
    dequeue  $u$  from  $Q$   
    for each node  $w$  adjacent to  $u$  do  
      if  $\text{mark}[w] \neq \text{visited}$  then  $\text{mark}[w] \leftarrow \text{visited}$   
      enqueue  $w$  into  $Q$ 
```

In both cases we need a main program to start the search.

```
procedure search( $G$ )  
  for each  $v \in N$  do  $\text{mark}[v] \leftarrow \text{not-visited}$   
  for each  $v \in N$  do  
    if  $\text{mark}[v] \neq \text{visited}$  then  $\{\text{dfs2 or bfs}\}(v)$ 
```

Fouille en largeur

Utilité

- Trouvera le sommet recherché, dans un graphe infini de degré borné, si un tel sommet existe.

Fouille d'un graphe par retour arrière (backtracking)

B&B Section 9.6

Contexte :

- graphe souvent implicite, car trop grand ou même infini
- souvent sans cycle, même un arbre
- sommet = solution partielle
- recherché : sommet qui est solution complète.

L'idée : étendre constamment une solution partielle et rebrousser chemin dès la détection de l'absence de solution complète le long d'une branche.

Retour arrière

Forme générale

```
procedure backtrack( $v[1..k]$ )  
  {  $v$  is a  $k$ -promising vector }  
  if  $v$  is a solution then write  $v$   
  { else } for each  $(k + 1)$ -promising vector  $w$   
    such that  $w[1..k] = v[1..k]$   
    do backtrack( $w[1..k + 1]$ )
```

Note : w n'est pas "weight" mais simplement un vecteur qui prolonge v .

Retour arrière : exemple 1

SAC À DOS AVEC MULTIPLICITÉS

DONNÉE: capacité $W \in \mathbb{R}^{\geq 0}$ et **types** d'objets $1, 2, \dots, n$ de poids $w_1, \dots, w_n \in \mathbb{R}^{\geq 0}$ et de valeurs $v_1, \dots, v_n \in \mathbb{R}^{\geq 0}$

CALCULER: comme d'habitude mais avec les $x_i \in \mathbb{N}$

Le retour arrière ressemble à la fouille en profondeur :

```
function backpack( $i, r$ )  
    {Calculates the value of the best load that can  
     be constructed using items of types  $i$  to  $n$   
     and whose total weight does not exceed  $r$ }  
     $b \leftarrow 0$   
    {Try each allowed kind of item in turn}  
    for  $k \leftarrow i$  to  $n$  do  
        if  $w[k] \leq r$  then  
             $b \leftarrow \max(b, v[k] + \textit{backpack}(k, r - w[k]))$   
    return  $b$ 
```

Appel initial : $\textit{backpack}(1, W)$.

Objets 1,2,3,4 de valeurs 3,5,6,10 et poids 2,3,4,5, capacité 8.
 Arbre typique d'un algo de retour arrière :

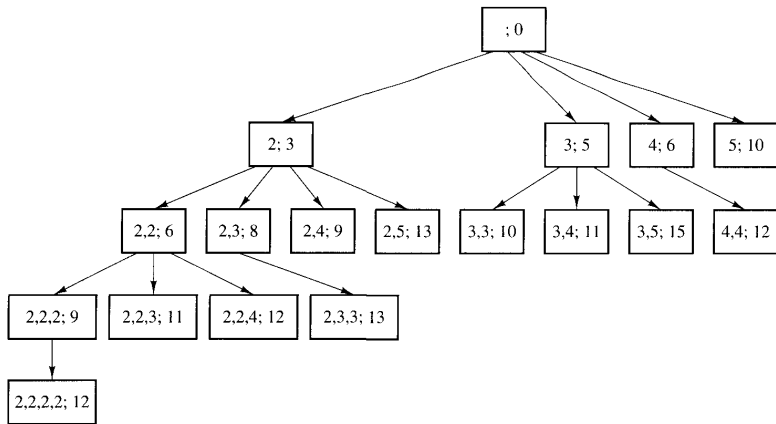


Figure 9.12. The implicit tree for a knapsack problem

2, 2, 4; 12 représente objets 1,1,3 (poids 2 et 2 et 4) totalisant valeur 12

Retour arrière : exemple 2

Peut-on placer 8 reines sur le jeu sans que 2 reines ne soient en prise ?

En classe.

Fouille par “séparation et évaluation” (branch and bound)

B&B Section 9.6

Contexte : chaque sommet est solution mais on cherche l'optimale.

L'idée : estimer pour chaque sommet visité une valeur de “favorabilité” et explorer ensuite les branches paraissant les plus favorables.

- raffinement du retour-arrière
- programmation inélégante car ni en profondeur, ni en largeur
- difficile et souvent impossible à analyser de manière théorique.

Principe du minimax

B&B Section 9.8

Contexte :

- graphe implicite d'un jeu à deux joueurs (ex : échecs)
- sommet = configuration du jeu (ex : positionnement des pièces)
- arc $s_1 \rightarrow s_2$ = coup possible de s_1 vers s_2
- chaque s reçoit une valeur $v(s)$ de “favorabilité envers le joueur A ”
- recherché : un bon coup de A partant de s
- heuristique seulement car $v(s)$ imparfaite.

Principe du minimax

(suite)

L'idée : un bon coup de A à partir de s est de jouer vers s_1 si

- $s \rightarrow s_1$ et

$$v(s_1) = \max_{s \rightarrow s'} \{v(s')\},$$

- ou mieux encore $s \rightarrow s_1 \rightarrow s_2$ et

$$v(s_2) = \max_{s \rightarrow s'} \{ \min_{s' \rightarrow s''} \{v(s'')\} \},$$

- ou mieux encore $s \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$ et

$$v(s_3) = \max_{s \rightarrow s'} \{ \min_{s' \rightarrow s''} \{ \max_{s'' \rightarrow s'''} \{v(s''')\} \} \}$$

- et ainsi de suite selon puissance de calcul disponible !