INF4705 — Analyse et conception d'algorithmes

CHAPITRE 7 : ALGORITHMES DE PARCOURS DE GRAPHE

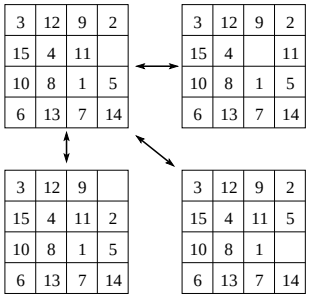
Motivation :

De nombreux problèmes sont représentés à l'aide d'un graphe. Pour les résoudre, on a souvent besoin de parcourir ce graphe. Il existe quelques algorithmes génériques de parcours de graphe, chacun avec ses caractéristiques que nous pouvons exploiter.

- Dans certains cas le graphe est représenté explicitement (p.ex. plus court chemin, arbre sous-tendant minimum).
- Dans d'autres cas le graphe est de trop grande taille pour qu'on souhaite le représenter au complet en mémoire ; à tout moment, on n'en manipule qu'une petite partie (graphe implicite). Par exemple pour résoudre des problèmes combinatoires, l'ensemble des configurations (partielles) possibles et les relations entre elles forment typiquement un très gros graphe.

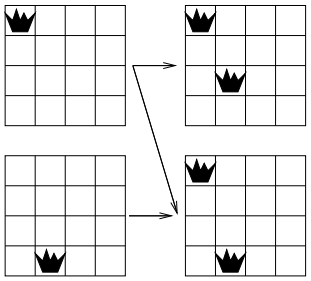
Exemple 1 *Configurations totales : Jeu de Taquin*

Déplacer les tuiles en les glissant afin de les placer en ordre numérique.



Exemple 2 *Configurations partielles : 4-reines*

Formulation générale : Placer n reines sur un échiquier de taille $n \times n$ de telle façon qu'elles ne se menacent pas ($\neq$ ligne, colonne et diagonale).



Nous rappellerons les techniques générales de parcours de graphe puis nous donnerons des exemples d'algorithmes les utilisant.

7.1 Parcours de graphes

7.1.1 Fouille en profondeur

S'exprime facilement de façon récursive ;
voici une formulation non récursive, pour fin de comparaison avec le prochain type de fouille :

procédure fep(v : nœud) {avec *pile* P}

1. P $\leftarrow \emptyset$;
2. visité[v] $\leftarrow$ vrai ;
3. **empiler**(v,P) ;
4. **tant que** P $\neq \emptyset$ **faire**
   (a) **tant que** $\exists w$ adjacent à **haut**(P) t.q. non visité[w] **faire**
      i. visité[w] $\leftarrow$ vrai ;
      ii. **empiler**(w,P) ;
   (b) **dépiler**(P) ;

7.1.2 Fouille en largeur

Visite d'abord tous les nœuds adjacents au nœud courant avant de passer au reste du graphe.

procédure fel(v : nœud) {avec *file* F}

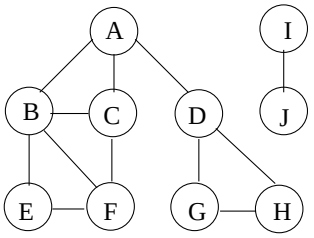
1. F $\leftarrow \emptyset$;
2. visité[v] $\leftarrow$ vrai ;
3. **mettre**(v,F) ;
4. **tant que** F $\neq \emptyset$ **faire**
   (a) **tant que** $\exists w$ adjacent à **premier**(F) t.q. non visité[w] **faire**
      i. visité[w] $\leftarrow$ vrai ;
      ii. **mettre**(w,F) ;
   (b) **sortir**(F) ;

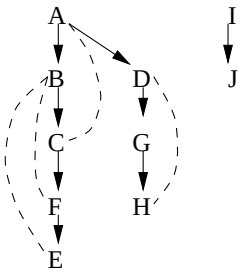
7.1.3 Et si le graphe n'est pas connexe ?

procédure parcours($G = (N, A)$: graphe, fouille : fep ou fel)

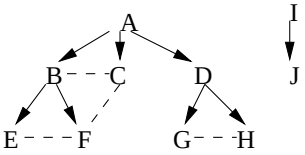
$\forall v \in N$ **faire**
  visité[v] $\leftarrow$ faux ;
$\forall v \in N$ **faire**
  si non visité[v] **alors** fouille(v) ;

Exemple 3 *graphe non-orienté*





fep: A,B,C,F,E,D,G,H,I,J



fel: A,B,C,D,E,F,G,H,I,J

Voici ce que contiennent la pile et la file en cours d'exécution :
P : *A ;AB ;ABC ;ABCF ;ABCFE ;A ;AD ;ADG ;ADGH ; ;I ;IJ ; ;*
F : *A ;BCD ;CDEF ;DEF ;EFGH ;FGH ;GH ;H ; ;I ;J ; ;*

Remarques :
- fep et fel prennent un temps dans $\Theta(\max(n, a))$: un nœud est mis une seule fois sur la pile/file ; une arête est considérée deux fois.
- On constate dans la littérature scientifique qu'il y a davantage d'algorithmes conçus à partir d'une fep qu'à partir d'une fel. Ceci est dû fondamentalement au fait qu'une fep offre la possibilité supplémentaire de traiter un nœud v après qu'on ait découvert les nœuds qu'on peut atteindre à partir de v (autrement dit en post-ordre), ce qui donne potentiellement plus d'information et permet d'effectuer un traitement plus sophistiqué.
- Classification des arêtes selon la première fois qu'on les rencontre :
  - Les arêtes utilisées par la fouille forment une forêt. Nous les appelerons *arêtes d'arbre*.
  - Pour une fep dans un graphe non orienté, toutes les autres arêtes sont des *arêtes arrière* : elles joignent un nœud à un de ses ancêtres.
  - Pour une fel dans un graphe non orienté, toutes les autres arêtes sont des *arêtes latérales* : elles joignent deux nœuds sans lien de descendance.
  - Pour une fep dans un graphe orienté, nous pouvons avoir des arcs d'arbre, arrière, latéraux et *avant* : d'un nœud vers un de ses descendants.
  - Pour une fel dans un graphe orienté, nous pouvons avoir des arcs d'arbre, arrière et latéraux.

Comment faire pour savoir si un graphe non orienté G :
- est connexe ou sinon pour connaître ses composantes connexes ?
  parcours(G,fep) ou parcours(G,fel) : si on obtient un seul arbre, le graphe est connexe ; sinon les arbres sont les composantes connexes.
- contient un cycle, en un temps dans O(n) ?
  Si on a $\geq n$ arêtes, on a nécessairement un cycle.
  Sinon

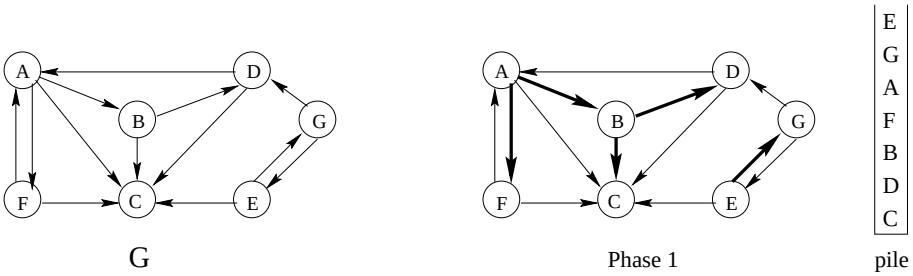
  parcours(G,fep) : cycle ssi $\exists$ arête arrière.

  parcours(G,fel) : cycle ssi $\exists$ arête latérale.

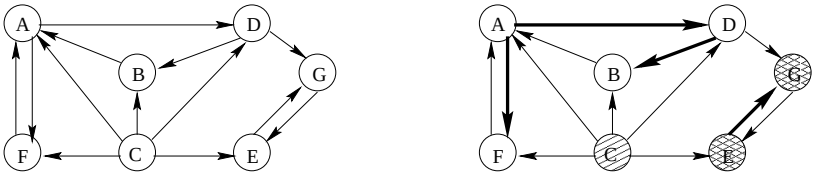
Comment faire pour savoir si un graphe orienté G :
- contient un cycle ?
  parcours(G,fep ou fel) : cycle ssi $\exists$ arête arrière. (Comment déterminer efficacement s'il s'agit d'une arête arrière ?)
- est fortement connexe ou sinon pour connaître ses composantes ?

1. Appliquer parcours(G,fep) et placer les nœuds dans une pile P selon le post-ordre.
2. Appliquer parcours(G^T,fep) ($G^T \equiv G$ avec ses arcs inversés) mais en utilisant P pour lancer les appels à fep. Chaque arbre obtenu correspond à une composante fortement connexe.

Exemple 4



G Phase 1



G^T Phase 2

7.2 Parcours d'arbres

Les trois parcours suivants sont un raffinement de la fouille en profondeur pour le cas particulier où le graphe est un arbre :

7.2.1 pré-ordre

On visite le nœud lui-même, puis récursivement les sous-arbres de ses enfants.

procédure pré-ordre(u : nœud)
    {Soient $v_1, \ldots, v_k$ les enfants de u}
    visiter u ;
    pour $i \leftarrow 1$ **à** k **faire**
        pré-ordre(v_i) ;

7.2.2 post-ordre

On visite d'abord récursivement les sous-arbres de ses enfants, puis le nœud lui-même.

procédure post-ordre(u : nœud)
    {Soient $v_1, \ldots, v_k$ les enfants de u}
    pour $i \leftarrow 1$ **à** k **faire**
        post-ordre(v_i) ;
    visiter u ;

7.2.3 en-ordre

On visite récursivement le sous-arbre de son premier enfant, ensuite le nœud lui-même, puis récursivement les sous-arbres de ses autres enfants.

procédure en-ordre(u : nœud)
  {Soient $v_1, \ldots, v_k$ les enfants de u}
  si $k = 0$ {u est une feuille}
    alors visiter u ;
    sinon
      en-ordre(v_1) ;
      visiter u ;
      pour $i \leftarrow 2$ **à** k **faire**
        en-ordre(v_i) ;

7.2.4 par niveau

Si on applique une fouille en largeur à un arbre, on obtient un parcours par niveau.

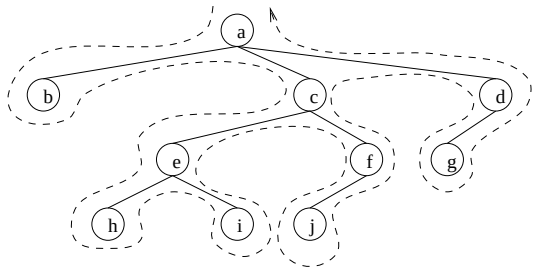
7.2.5 Détermination facile des parcours

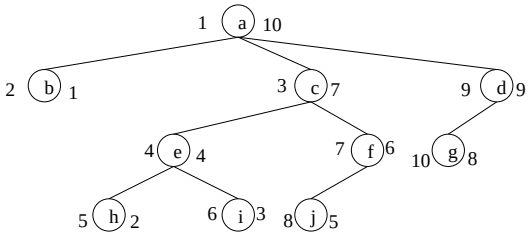
Truc pour les simples mortels : On fait une promenade autour de l'arbre en partant de la racine et se dirigeant d'abord vers le premier enfant. L'ordre de rencontre de certaines fenêtres aux nœuds nous donne le parcours désiré.

pré-ordre : fenêtre ouest ; a,b,c,e,h,i,f,j,d,g

post-ordre : fenêtre est ; b,h,i,e,j,f,c,g,d,a

en-ordre : fenêtre sud ; b,a,h,e,i,c,j,f,g,d



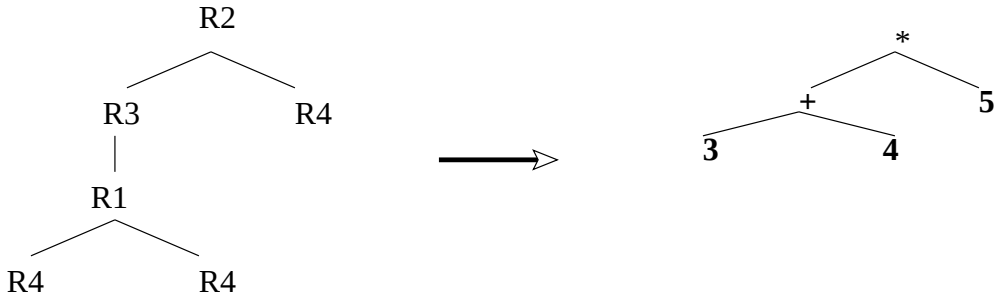


7.3 Évaluation d'expressions arithmétiques

Voici un extrait d'une grammaire qu'un compilateur pourrait utiliser pour traiter une expression arithmétique :

R1 Expr → Expr + Expr
R2 Expr → Expr × Expr
R3 Expr → (Expr)
R4 Expr → Nombre

L'expression arithmétique peut être représentée par un arbre dont les feuilles sont des opérandes et les nœuds internes, des opérateurs. Ainsi, il n'est plus nécessaire d'avoir recours aux parenthèses. Par exemple,



représente l'expression $(3 + 4) \times 5$.

Un parcours en post-ordre de l'arbre de droite nous donne : $3\ 4 + 5\ \times$. L'algorithme d'évaluation utilise une pile et un parcours post-ordre de l'arbre :

- Chaque fois qu'on visite une feuille, on place la valeur correspondante dans la pile.
- Chaque fois qu'on visite un nœud interne, on retire de la pile autant de valeurs que l'arité de l'opérateur correspondant (donc deux valeurs pour notre exemple puisqu'il s'agit d'opérateurs binaires), on effectue l'opération puis on place le résultat dans la pile.
- À la fin du parcours, le résultat final se retrouve dans la pile.

7.4 Détermination d'ancêtres

Étant donné deux nœuds u et v dans l'arbre, u est-il un ancêtre de v ?
u est *ancêtre* de v s'il apparaît sur le chemin de la racine à v.

Nous ferons un pré-traitement de l'arbre en temps et espace linéaire qui nous permettra de répondre à de telles questions en temps constant.

Numérotons les nœuds selon un parcours en pré-ordre (prénum(u)) et en post-ordre (postnum(u)).

Remarques :

1. prénum(u) $\leq$ prénum(v) $\Leftrightarrow$ (u est un ancêtre de v) $\vee$ (u est à la gauche de v dans l'arbre)
2. postnum(u) $\geq$ postnum(v) $\Leftrightarrow$ (u est un ancêtre de v) $\vee$ (u est à la droite de v dans l'arbre)
3. Donc (prénum(u) $\leq$ prénum(v)) $\wedge$ (postnum(u) $\geq$ postnum(v)) $\Leftrightarrow$ u est un ancêtre de v

7.5 Couplage maximum dans un graphe biparti

Exemple 5 *Les mariages arrangés*
Étant donné un groupe d'hommes et un autre de femmes, nous voulons arranger le plus grand nombre de mariages possible, sachant que certains couples ne sont pas permis. Ceci correspond à la recherche d´un couplage *maximum dans un graphe biparti.*

L'algorithme de couplage est basé sur le concept de chemin augmentant. Étant donné un certain couplage C dans le graphe, un chemin augmentant *relie deux nœuds non couplés dans C et est composé en alternance d'arêtes faisant partie et ne faisant pas partie de C. Un tel chemin est nécessairement de longueur impaire.* Pourquoi ?

Théorème 1 *Un couplage est maximum ssi il n'existe pas de chemin augmentant.*

L'algorithme est donc le suivant :

fonction *couplageMax($G = (X, Y, A)$: graphe biparti) : couplage*

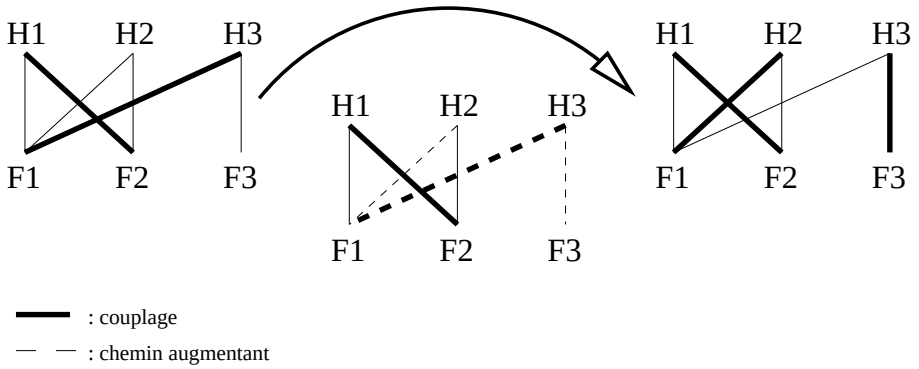
$C \leftarrow \emptyset$;

tant que $\exists$ *chemin augmentant, disons* γ, **faire**

$C \leftarrow C \oplus \gamma$;

retourner C

Pour trouver un chemin augmentant, s'il en existe un, on applique une fel à partir des nœuds non couplés jusqu'à ce qu'on trouve un autre nœud non couplé à une distance impaire ou qu'on soit bloqué.



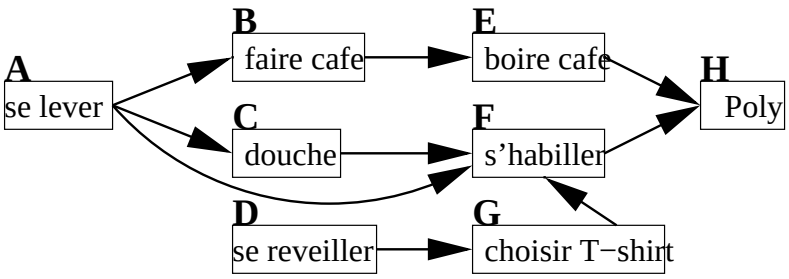
Analyse : Le temps de calcul de cet algorithme est dans $\mathcal{O}(n^3)$ puisqu'on effectuera au plus $\frac{n}{2}$ itérations (chaque chemin augmentant ajoute une arête au couplage ; on ne peut avoir plus de $\frac{n}{2}$ arêtes dans le couplage) prenant chacune un temps dans $\Theta(\max(a, n))$.

7.6 Tri topologique

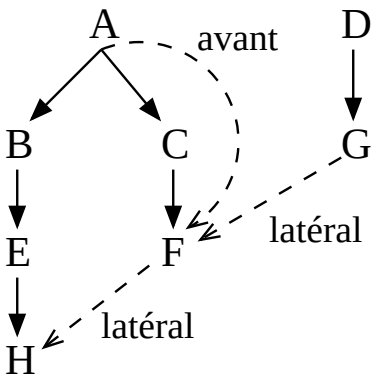
Étant donné un graphe orienté sans cycle, ordonner ses nœuds tel que $\forall$ arc (u, v), u précède v.

On peut modifier l'algorithme fep en ajoutant "4(a-bis) **écrire haut**(P) ;" (équivalent à un parcours post-ordre de l'arbre engendré). Ceci donne l'ordre voulu mais en sens inverse.

Exemple 6 *Prenons un problème de tous les jours...*



Fouille en profondeur de ce graphe :



On obtient par exemple D G A C F B E H mais d'autres solutions existent aussi (comme A B C D E G F H).

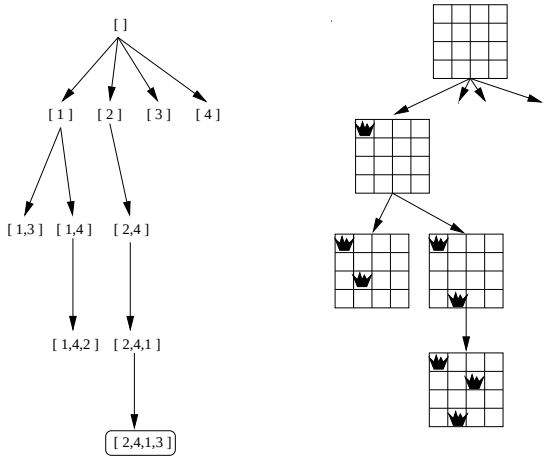
7.7 Fouille à retour arrière ("backtracking")

Il s'agit essentiellement d'une fep sur un graphe implicite.

Exemple 7 *Les 4-reines*
On ne peut pas avoir deux reines dans la même colonne donc nous pouvons représenter une solution par un vecteur $[\ell_1, \ldots, \ell_4]$ *indiquant la ligne sur laquelle se trouve chacune des reines.*

Un vecteur $[\ell_1, \ldots, \ell_k]$, $0 \leq k \leq 4$ *est une* configuration (partielle) légale *si aucune des* k *reines ne se menace.*

On définit un graphe orienté dont les nœuds sont les configurations légales et où il y a un arc de $[\ell_1, \ldots, \ell_k]$ *à* $[\ell_1, \ldots, \ell_k, \ell_{k+1}]$.



Remarque : Si on désire la solution nécessitant le moins d'étapes, on peut vouloir utiliser une fel, par exemple pour le Jeu de Taquin ou le cube Rubik.

7.8 Séparation et évaluation progressive ("branch-and-bound")

Dans le contexe d'un problème d'optimisation, on effectue une fouille à retour arrière avec en plus à chaque nœud le calcul d'une borne sur la valeur de toutes les solutions potentielles dans le sous-arbre dont ce nœud est la racine.

Nous nous servons de la borne pour éliminer des sous-arbres inattrayants dans notre arbre de recherche. Nous pouvons aussi nous en servir pour guider l'exploration de cet arbre.

Exemple 8 *Sac à dos*
On veut maximiser

$$\Sigma_{i=1}^{n} x_i v_i, \quad x_i \in \{0,1\}$$

tout en respectant $\Sigma_{i=1}^{n} x_i w_i \leq W$.
Supposons sans perte de généralité que les objets sont considérés en ordre lexicographique.

Borne A : *Lorsque* $x_1, \ldots, x_k$ *sont fixées, notre borne supérieure sera :*

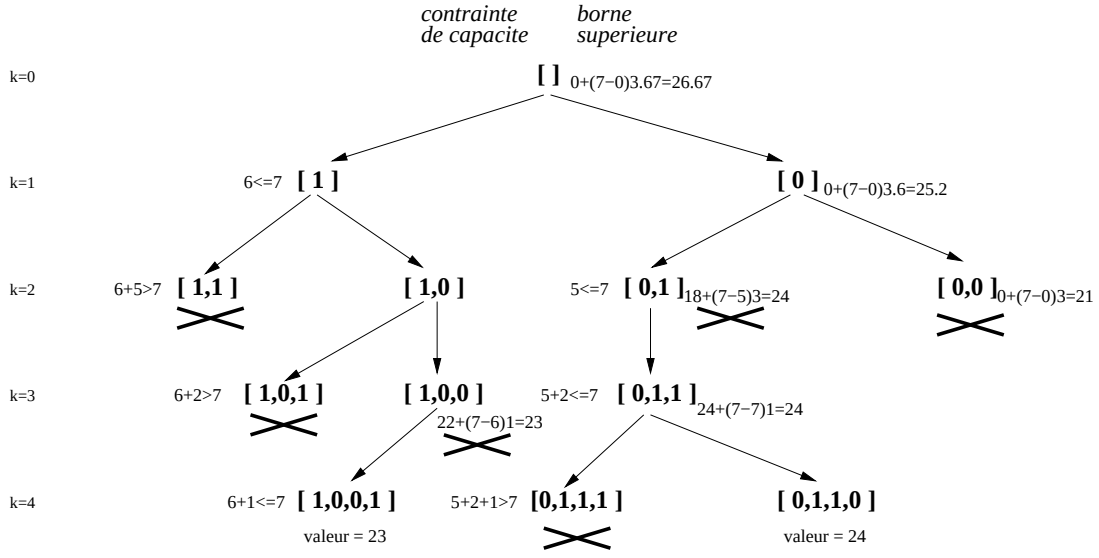
$$\Sigma_{i=1}^{k} x_i v_i + (W - \Sigma_{i=1}^{k} x_i w_i) \cdot v_j / w_j, \quad \text{où } j = argmax_{k+1 \leq i \leq n} v_i / w_i$$

Borne B : *En relâchant la contrainte d'intégralité,* $x_i \in \{0,1\}$ *devient* $0 \leq x_i \leq 1$ *pour* $k+1 \leq i \leq n$. *La valeur de la solution optimale de cette relaxation du problème, obtenue par exemple avec un algorithme vorace utilisant la densité de valeur comme critère de choix, est au moins aussi grande que celle de la solution optimale au problème originel. C'est une meilleure borne que la précédente puisque borne* $B \leq$ *borne* A.

Les nœuds de notre graphe implicite sont les configurations légales $[x_1, \ldots, x_k]$, $0 \leq k \leq n$, *où "légale" signifie ici qu'on ne dépasse pas la capacité* W.

Reprenons un exemple déjà rencontré, avec $n = 4$, $W = 7$. *Considérons les objets en ordre non croissant de densité de valeur,* v_i/w_i, *et utilisons la borne A.*

objet	*valeur*	*poids*	v_i/w_i
1	*22*	*6*	*3.67*
2	*18*	*5*	*3.60*
3	*6*	*2*	*3.00*
4	*1*	*1*	*1.00*



Une solution optimale est donc le choix des objets 2 *et* 3*, pour une valeur totale de* 24*. Treize nœuds ont été générés sur un total possible de* $\Sigma_{i=0}^{4} 2^i = 2^{4+1} - 1 = 31$.

Nous n'avons fait qu'effleurer le sujet. Il y aurait plusieurs sources d'amélioration possibles :

"best-first" : plutôt que de développer l'arbre en profondeur d'abord, on pourrait choisir de le développer là où la borne est la plus optimiste (meilleur d'abord) mais ceci requiert qu'on garde en mémoire la frontière de l'arbre (on fait une fel), qui croît rapidement ;
[Note : Un "best-first" jumelé à une heuristique *admissible* (borne inférieure du nombre d'étapes restantes) correspond à l'algorithme $A^\star$ en IA pour la recherche d'un plus court chemin d'un état initial à un état cible. Pour le Jeu de Taquin, le "nombre de tuiles qui ne sont pas à leur place" est une heuristique admissible ; la "somme des distances manhattan de chacune des tuiles à sa position finale" est en général un meilleur guide mais n'est pas admissible.]

bornes : le calcul de bornes peut être beaucoup plus sophistiqué et est parfois basé sur une relaxation de certaines contraintes du problème (comme la borne B) ;

heuristiques de branchement : d'autres critères peuvent intervenir pour décider de l'ordre dans lequel les candidats sont considérés dans l'arbre ou encore de l'ordre d´exploration des branches à un nœud donné ;

anticipation/inférence : on peut réduire la taille du sous-arbre à un nœud en anticipant les impasses ("forward-checking", "look-ahead", techniques de cohérence).