Master d'Informatique - Spécialité Androide
Module MAOA

Recherche Opérationnelle et Optimisation Combinatoire

(Rappels) Branchement et Evaluation (Branch-and-Bound)

Pierre Fouilhoux

Sorbonne Université

2019-2020

1. Illustration par le problème du voyageur de commerce
2. Cadre classique du Branch-and-Bound en PLNE
3. Analyse fine de la combinatoire
4. Efficacité d'un algorithme de branchement

Considérons le problème du voyageur de commerce afin d'introduire la méthode de Branch&Bound.

Considérons $n+1$ villes $v_0, v_1.., v_n$
et les distances inter-villes d_{ij}, $i, j \in \{0, .., n\}$.
Le problème du voyageur de commerce consiste à donner une tournée allant et revenant à la ville v_0 en passant une fois par chacune des villes.

Une solution de ce problème s'écrit donc sous la forme d'une liste de villes données dans l'ordre de la tournée, que l'on peut noter $\sigma(1), ..., \sigma(n)$ où σ est une permutation des n villes.

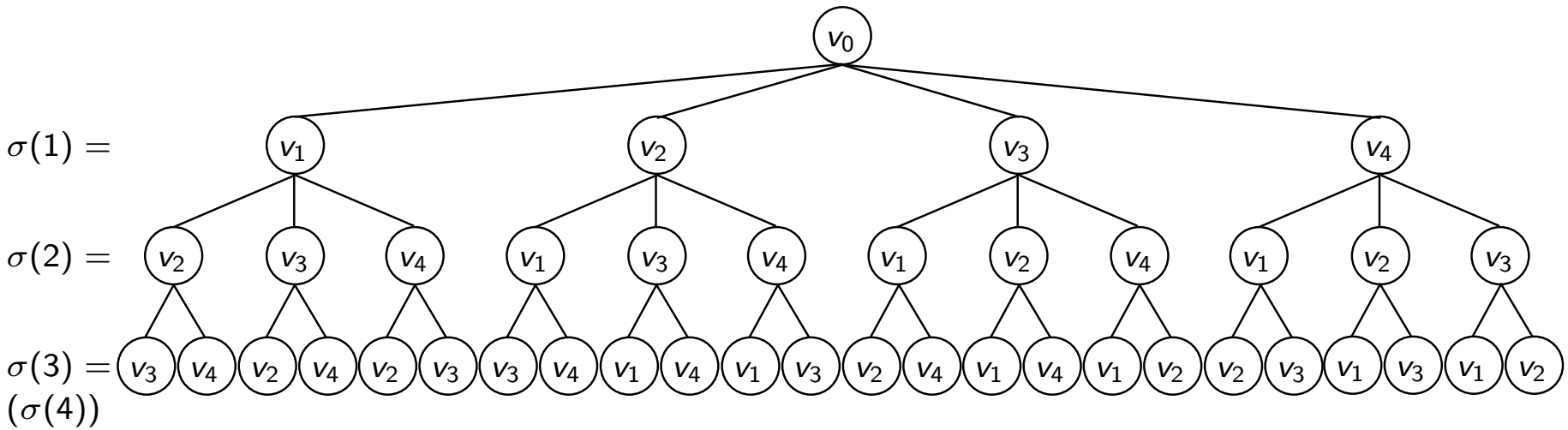
On a donc clairement $n!$ solutions possibles. Nous avons vu que l'explosion combinatoire d'une énumération la rend difficilement envisageable dans tous les cas. Néanmoins, dans cette section, nous nous intéressons aux méthodes énumératives, c'est-à-dire des méthodes qui explorent l'espace des solutions pour trouver une solution de meilleur coût.

Les méthodes de Branch&Bound essayent d'éviter d'explorer entièrement l'espace des solutions en utilisant les caractéristiques des solutions optimales et des estimations du coûts des solutions. Malheureusement, ces améliorations ne changent pas la complexité de ces méthodes qui restent de complexité exponentielle. C'est pourquoi il est souvent nécessaire de chercher d'autres techniques de résolution. Ce que nous verrons dans les chapitres suivants.

Prenons dans cet exemple 5 villes v_0, v_1, v_2, v_3 et v_4 et les distances inter-villes données par le tableau suivant :

	v_0	v_1	v_2	v_3	v_4
v_0	-	8	4	2	3
v_1	9	-	7	1	6
v_2	3	7	-	6	6
v_3	2	1	6	-	4
v_4	7	6	6	2	-

Pour construire une méthode énumérative, il faut choisir une façon d'énumérer les solutions de façon à n'en oublier aucune et en évitant de traiter plusieurs fois la même solution. Dans notre exemple, on peut considérer l'arborescence d'énumération suivante.



A chaque niveau de l'arborescence en partant de la racine, on fixe quelle ville sera visitée à la $i^{\text{ème}}$ place de la tournée ($i = 0$, puis $i = 1$, puis $i = 2$,...). Ainsi, chaque nœud correspond à une tournée partielle (et valide) des villes. Par exemple, le $4^{\text{ème}}$ nœud en partant de la gauche du niveau où l'on affecte la $2^{\text{ème}}$ place, correspond à la tournée partielle $\{v_0, v_2, *, *\}$. Au rang le plus bas, chaque feuille correspond à une tournée complète, c'est-à-dire à une solution du problème.

Notre but est donc de décider si l'exploration d'une branche est utile avant d'en énumérer toutes les solutions. Pour cela, on va donner une *évaluation* des solutions que peut porter une branche, c'est-à-dire donner une borne inférieure des coûts des solutions de la branche. Si cette évaluation est plus grande qu'une solution que l'on connaît déjà, on peut *élaguer* cette branche, c'est-à-dire éviter de l'explorer.

Dans notre exemple du voyageur de commerce, avant de commencer l'énumération, tentons de trouver une bonne solution.

Par exemple, on peut prendre une heuristique gloutonne qui consiste à prendre en premier la ville la plus près de v_0, puis en deuxième une ville (parmi les villes restantes) qui soit la plus près de la première, et ainsi de suite. On obtient ainsi une solution réalisable $s_1 = \{v_0, v_3, v_1, v_4, v_2\}$ de coût $c_1 = 18$. Une solution réalisable nous donne une bonne supérieure pour notre problème, en effet, nous savons que la solution optimale du problème est au plus de coût 18.

Il nous serait à présent utile de trouver une évaluation du coût de la solution par une borne inférieure.

En effet, si l'on pouvait par exemple déterminer que la solution optimale est au moins de coût 18, nous aurions déjà résolu notre problème. Une idée possible ici est de dire que l'on passe par toutes les villes et que l'on doit donc sortir de chacune des villes pour aller vers une autre ville. En d'autre terme, le trajet pour aller d'une ville v_i à une autre ville sera au moins égale à la plus petite distance issue de v_i.

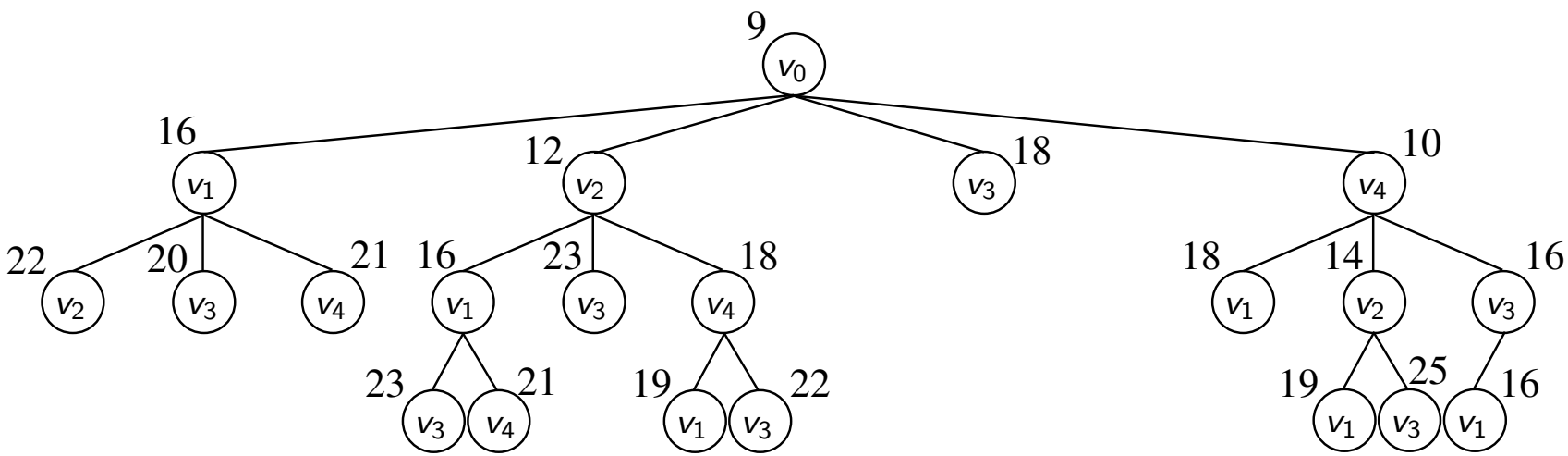
Ainsi, en sommant les coûts minimaux de chaque ligne du tableau, on obtient une valeur qui sera toujours inférieure au coût d'une solution du problème. Par exemple, ici on obtient une évaluation de 2+1+3+2+2=9. Une solution de ce problème sera donc au moins de coût 9. Malheureusement, la meilleure solution que nous connaissons est de coût 18, nous devons donc commencer l'énumération.

La première branche à gauche issue de la racine de l'arborescence contient toutes les solutions où v_1 est affectée à la place 1.

Si l'on applique notre idée d'évaluation à la sous-matrice obtenue en ôtant la ligne correspondant à v_0 (on connaît déjà la ville qui suit v_0 et la colonne correspondant à v_1 (la ville allant en v_1 est déjà fixée, c'est v_0), on obtient alors 8 pour la somme des coûts minimaux des lignes. En ajoutant le coût $c_{v_0 v_1} = 8$, on sait alors que toute solution de cette branche a au moins une coût de 16.

On explore alors le noeud de l'arborescence commençant par $\{v_0, v_1, v_2\}$. Cela correspond à une matrice où l'on a ôté les lignes correspondant à v_0 et v_1 et les colonnes correspondant à v_1 et v_2. La somme des coûts minimaux des lignes restantes est alors de 7. En ajoutant alors le coût $c_{v_0 v_1} + c_{v_1 v_2} = 15$, on sait que toute solution de cette branche est au moins de 22. Comme nous connaissons déjà une solution de coût 18, on peut donc éviter d'explorer (élaguer) cette branche.

La figure suivante résume l'exploration complète de notre arborescence par un parcours en profondeur. Les valeurs qui accompagnent les nœuds sont les évaluations des branches et celles qui accompagnent les feuilles sont les coûts des solutions associées.



Lors de cette exploration, on a pu élaguer totalement la $3^{\text{ème}}$ branche. C'est la dernière branche qui nous a conduit à une feuille correspondant à une solution $s_2 = \{v_0, v_4, v_3, v_1, v_2\}$ de coût 16. Or, la dernière branche ne peut pas mener à une solution de meilleur coût que 16, on a pas besoin d'explorer cette dernière branche. En conclusion, la meilleure solution pour le problème est de coût 16 et nous connaissons s_2 une des solutions optimales possibles.

Principe de branchement (Branch&Bound)

Nous considérons ici les algorithmes de Branch&Bound
dans le contexte d'une Maximisation :
on recherche une solution maximale optimale x dans $\{1, ..., k\}^n$.
Pour une minimisation, il suffit de remplacer tous les maxima par des minima et inversement.

Plutôt qu'énumérer, on utilise :
- le paradigme informatique classique du "diviser pour régner" (divide&conqueer) consistant à diviser le problème en sous-problèmes plus simples à résoudre.
- en tentant d'éviter l'exploration du plus possible de sous-problèmes.

Arborescence de branchement

- Problème "racine" : C'est la relaxation du PLNE à résoudre.

- Sous-problèmes : La résolution d'un problème complexe peut exiger de découper l'espace de ses solutions de manière à remplacer ce problème par plusieurs **sous-problèmes** ou **nœuds** (ou problèmes-fils) : ces sont des problèmes construits à partir du problème racine avec chacun un espace de solutions plus réduits que le problème racine.

- Arborescence : Le problème racine et les sous-problèmes générés produisent une arborescence à partir du sommet racine sur des nœuds fils.

"Brancher"

- "Brancher" : diviser un problème en un ensemble de sous-problèmes tels que l'union de leurs espaces de solutions forme l'espace des solutions du problème-père.

On appelle parfois le branchement une *séparation* mais ce nom est dangereux car confondu avec la séparation des algorithmes de coupes.

Le branchement le plus courant consiste à choisir l'une des variables x_i puis à définir un problème-fils pour chaque valeur de x_i prise dans $\{1, ..., k\}$.
Il est souvent utile de considérer des branchements plus généraux où l'on branche sur des sous-problèmes plus larges ($x_1 <= 3\,;\, x_1 > 3$) par exemple
ou même sur des contraintes plus complexes ($x_1 + x_2 <= 5\,;\, x_1 + x_2 > 5$).

On préfère bien entendu des sous-problèmes fils dont les espaces de définitions ont une partition de l'espace de leur père : cela permet de ne pas explorer deux fois une solution. Néanmoins, cela n'est pas nécessaire.

Evaluation

- Evaluer un nœud : trouver une borne supérieure de la valeur optimale (entière) d'un sous-problème.
Il est important de remarquer qu'alors cette évaluation est également une évaluation de tous les sous-problèmes issus de ce sous-problème !

- Relaxation : on appelle ici **relaxer un PLNE** le fait de résoudre un problème "plus simple" à la place du PLNE de manière à obtenir une évaluation du nòeud.

La plus "simple" des relaxations est la **relaxation linéaire** qui consiste à relaxant les contraintes d'intégrité pour obtenir un PL (mais il y a en d'autres (voir chapitres correspondants).
La relaxation linéaire fournit ainsi une borne supérieure sur toutes les solutions du nœud en temps polynomial.

- Borne supérieure (globale) : la plus grande des évaluations parmi tous les noeuds d'un même niveau de l'arborescence.

Ainsi l'évaluation du noeud racine est une borne supérieure à tous les nœuds.
On doit attendre d'avoir évaluer tous les nœuds d'un niveau de l'arborescence pour avoir une nouvelle borne supérieure.

"Stériliser"

- Nœud stérile (ou feuille) : un nœud qui ne produit pas de sous-problème lors du branchement :
    - soit car il est vide de solution (infaisable)
    - soit car il est entier

Ici le mot "entier" correspond au fait que la relaxation linéaire du nœud fournit une solution entière : comme la relaxation fournit une borne supérieure qui est alors atteinte, cette solution est la meilleure solution du sous-problème en version entière (il ne sert à rien de brancher à nouveau).

Le branchement arrive toujours à des nœuds entiers lorsqu'il se déroule jusqu'aux feuilles, car elles sont entières par définitions !

Solutions et borne inférieure

- Optimum local : si un nœud n'est pas stérile et qu'on en connaît la meilleure solution, on a alors une **solution associée** à ce nœud qui est un **optimum local** de ce sous-problème).

- Borne inférieure globale : la meilleure des solutions (entières) rencontrées au cours de l'exploration constitue une borne inférieure pour le PLNE.

- Elagage : On dit qu'on **élague** un nœud (ou qu'on coupe) ainsi que toute la branche d'une arborescence qui en est issue lorsqu'on décide de ne pas l'explorer. Cela est possible en **si l'évaluation de ce sous-problème est inférieure à la borne inférieure globales**, c'est-à-dire la meilleure solution connue que l'on a mise de côté bien entendu.

- Arrêt anticipé : dans le pire des cas, l'algorithme énumère quand même toutes les solutions... il est donc exponentiel.
Mais :
- il est possible que la borne supérieure globale devienne égale à la borne supérieure globale (ou même à +0,99999 si la valeur de la solution est elle-même entière) : l'algorithme s'arrête alors en ayant trouvé une solution optimale).
- il est possible d'arrêter par action humaine l'algorithme en cours d'exploration : si une solution entière a été déterminée : on a alors un encadrement par garantie expérimentale de cette solution grâce à la borne supérieure. Il est à noter que plus on dispose de temps, plus cet écart (gap) se réduit.

Schéma de l'algorithme

On appelle **algorithme de Branch&Bound** ou *séparation-évaluation* (B&B) les algorithmes de branchement utilisant les notions de stérilité, borne inf/borne sup et élagage afin d'éviter l'exploration systématique de toute l'arborescence.

On note ici L la structure de données des sous-problèmes à explorer

(1) L$\leftarrow$ Problème correspondant au PMD
(2) Tant que L non vide faire
(3) Extraire un sous-problème P de L.
(4) Si évaluation de P est $\geq$ à la borne inf alors
(5) Créer les problèmes-fils P_i de P
(6) Pour chaque P_i faisables dont l'évaluation est $\geq$ à la borne inf faire
(7) Si on dispose d'une solution associée à P_i alors mise à jour borne inf.
(8) Sinon ajout de P_i dans L.
(9) Fin Pour.
(10) Fin Si.
(11) Fin Tant que.

Autres caractéristiques

- **Structure de données** L : il s'agit d'une simple liste de sous-problème : il n'y a pas d'arbres à proprement parlé.
  En revanche, un codage économique des sous-problèmes induit une arborescence. Ene effet, les sous-problème étant construits les une à partir des autres selon l'arborescence : l'essentiel des informations d'un sous-problème est redondante par rapport à son père (et ses frères) : la meilleure façon de stocker ces informations est une structure d'arborescence.
  Lors de l'élaguage (ligne 4), il peut aussi vouloir simultanément vider partiellement L de tous les problèmes d'évaluation strictement supérieure à la borne inf. Il existe de nombreuses astuces pour réduire l'espace mémoire nécessaire et accélérer les opérations sur les structures de données.

- **Choix du branchement** : La création de sous-problème (ligne 5) est souvent appelé le choix du branchement : il s'agit dans le cas le plus simple de choisir quelle variable sera utilisée pour brancher (dite alors variable de branchement).

Stratégie de branchement

L'ordre dans lequel sont traités les sous-problèmes stockés dans la structure L a une influence sur le fonctionnement de l'algorithme. On parle alors de **stratégie de branchements**.
Elle semble a priori importante pour une bonne exploration. Les stratégies classiques consistent soit à parcourir l'arborescence des sous-problèmes en largeur (breadth-first search), en profondeur (deep-first search) ou en recherchant le sous-problème non traité de meilleur évaluation (best-first search). Mais cela n'est pas toujours statistiquement vraie qu'une stratégie est meilleure qu'une autre.

Lorsque l'on dispose d'une fonction d'évaluation, il peut sembler judicieux d'opter pour la politique d'exploration du meilleur élément. On utilise alors une structure de tas pour coder L. Les solveurs utilisent en général une stratégie consistant à effectuer d'abord quelques descentes en profondeur dans l'arbre afin d'obtenir des bornes inf, puis une exploration par meilleure évaluation.

On peut noter que l'exploration en profondeur ne nécessite pratiquement pas de mémoire : en effet, l'exploration des branches les unes après les autres n'exigent pas de retenir tout l'arbre. On appelle alors l'algorithme de B&B l'algorithme *d'énumération implicite*.

Cadre "classique" d'un Branch-and-Bound pour la PLNE

- l'évaluation est obtenue par relaxation continue du PLNE.
- la stratégie de branchement consiste à effectuer un peu au hasard des explorations en profondeur de l'arbre afin de générer des solutions réalisables qui produiront des bornes inf (en maximisation). Ensuite, l'algorithme réalise une exploration par "meilleur noeud d'abord".
- le branchement est effectué sur les variables, en choisissant la variables la "plus fractionnaire". Par exemple, pour des variables binaire, on choisit la variable de valeur la plus proche de 0,5. En pratique, les solveurs branche plutôt sur la première variable "un peu" fractionnaire rencontrée.

Prétraitement

Il est important de noter que les différences de capacité entre les algorithmes génériques sur les PMD (c'est-à-dire les capacités des produit Cplex, Scip ou Gurobi) proviennent de bonnes implémentations des algorithmes de branchements, mais pas seulement.

Une bonne part de leurs capacités proviennent de techniques de **prétraitement** sur les variables et les contraintes. Ces prétraitements sont des algorithmes qui explorent la structure du problèmes pour rechercher par exemple des inférences numériques (dominances par exemple) ou logique (si les variables sont binaires) : ces algorithmes sont basées sur des moteurs de programmation par contraintes. Ces prétraitements permettent ainsi de réduire le nombre de variables et de contraintes : dans les problèmes "réels" tirés d'applications industrielles, ces prétraitement sont même parfois capables de réduire la taille des PMD de manière étonnante.

Au cours de l'algorithme, ce prétraitement permet également de réduire l'espace à explorer en utilisant des inférences logiques sur les variables/contraintes.

L'augmentation des capacités régulières depuis 2010 des solveurs de PLNE provient surtout de méthodes théoriques que nous étudieront dans les parties suivantes !

Analyse algorithmique "fine" de la combinatoire

Il est important d'analyser avec une grande précision la nature du problème combinatoire.

Bien entendu une étude de complexité pour ne pas utiliser un algorithme de branchement exponentiel sur un problème polynomial !

Mais, même dans le cas d'un problème NP-difficile, une étude fine peut être utile :

- repérer dans le problème ou au sein des instances à traiter s'il n'y a pas de prétraitement à effectuer.
- voir s'il n'y pas des propriétés de programmation dynamique qui peuvent rendre le problème pseudo-polynomial : le schéma de programmation dynamique peut alors être efficace pour les instances à résoudre.
- étudier s'il n'y a pas de propriétés de dominances de certaines solutions sur les autres pour réduire l'espace à explorer.

Programmation dynamique

La structure d'un problème permet fréquemment d'éviter une exploration systématique comme l'effectue un algorithme de B&B :

Lorsque les valeurs des solutions sont liées entre elles par une formule de récurrence basée sur le principe suivant. Supposons qu'une solution d'un problème est déterminé par une suite de décisions $D_1, D_2, ...D_p$: l'hypothèse dite de la **programmation dynamique** est qu'une prise de décisions optimales $D_1, ...D_p$ est telle que, pour tout entier $k = 1, ...p-1$, les décisions $D_{k+1}, ...D_p$ doivent être optimales. Cette hypothèse est très forte car elle a pour conséquence de donner un ordre à l'exploration des solutions en éliminant l'exploration de certaines. De plus, sous certaines conditions à prouver, elle permet même de montrer qu'il n'y a qu'un nombre polynomial de sous-problèmes (appelés alors *états*) à explorer.

On peut citer en exemple l'algorithme de Dijkstra pour les plus courts chemins ou la résolution des cas polynomiaux du problème du voyageur de commerce.

Pour un problème NP-complet, cette technique ne va bien entendu pas fournir un algorithme polynomial, mais elle peut amener à un algorithme **pseudo-polynomial** c'est-à-dire dont la complexité dépend de la valeur des paramètres et qui est polynomial pour une valeur donnée : c'est le cas par exemple du problème de sac-à-dos.

Dominances

Plus généralement Lorsque les solutions ont des structures fortement liées entre elles, on peut détecter des **dominances** entre solutions : si dans une arborescence, on peut montrer que tout sous-problème descendant d'un sous-problème a est au moins aussi bon que tout descendant d'un sous-problème b : on dit que a domine b. Dans ce cas, il suffit d'explorer a : on dit que a tue b (cela revient à stériliser b dès que a est explorer par exemple).

Ces dominances se rencontrent dans les problèmes à forte structure, par exemple des problèmes d'ordonnancement ou d'optimisation sur les graphes, où les solutions ont des points ou des parties en communs.

Cas particuliers polynomiaux et branchements

Une étude fine des cas particuliers du problème général peut être très utile :

- Il est possible que les instances traitées soient souvent dans une catégories simples que l'on peut traiter efficacement : si on peut en détecter la nature avant le traitement, on lance alors l'algorithme le plus simple (graphe sans circuit ou de degré limité, ordonnancement ayant une dominance sur les rapports poids/coûts,...).

- On peut aussi aiguiller le branchement de manière qu'à une certaine profondeur de l'arbre, un sous-problème ait les caractéristiques des instances "polynomiales". Ainsi lorsque que ce cas se présente, on résoud le sous-problème avec un outil polynomial (détection d'un graphe sans circuit, calcul du degré du graphe, etc).

Toute l'efficacité d'un algorithme de branchement repose en fait sur l'utilisation d'une comparaison entre des évaluations et minimums locaux et globaux. En effet :

- la meilleure des solutions valides rencontrées au cours de l'exploration constitue une borne inférieure.
- dans l'arborescence, si l'on considère la plus grande des évaluations (c'est-à-dire des bornes supérieures) d'un niveau, on obtient alors une borne supérieure pour le PMD que l'on appelle alors **borne supérieure globale**.

Ainsi, on peut agir sur l'exploration des sous-problèmes en **élaguant** certains sous-problèmes, c'est-à-dire que l'on explore pas ou que l'on stoppe l'exploration de la branche issue de ce problème car lorsque l'évaluation de ce sous-problème est inférieure à la meilleure solution connue pour le PMD entier.

L'autre aspect qui permet d'explorer moins de sous-problèmes est de permettre l'apparition de **nombreux sous-problèmes stériles**, ou plus précisemment de sous-problème pour lesquelles on connaît l'optimum local. Ceci peut s'obtenir par exemple en ayant une méthode d'heuristique pour produire ce minimum qui correspond à la borne supérieure locale.
On peut aussi atteindre des sous-problèmes correspondant à des PL ayant directement une solution entière : il faut pour cela avoir un PL entrant dans l'un des cas polynomiaux (voir sections suivantes).

Un autre point important de l'efficacité d'un algorithme est de pouvoir contourner le **problème des symétries.** Deux solutions sont symétriques si elles s'obtiennet l'une de l'autre en échangeant des valeurs des variables et si elles ont même coût. Un algorithme de branchement va devoir explorer toutes les solutions symétriques proches de l'optimum. Plusieurs techniques récentes permettent de contourner cette exploration : pertubations, branchement orbital,...

Outre une étude fine efficace, d'autres moyens d'améliorer le branchement sont possibles :

- **Comment faire pour qu'il y ait beaucoup de nœuds entiers AVANT les feuilles ?**
- **Comment faire pour avoir une meilleure évaluation (bornes supérieures) ?**
- **Comment faire pour avoir une meilleure borne inférieure (solution résalisables) ?**
- **Comment faire pour limiter l'explosion due aux symétries ?**

On va tenter de voir cela dans la suite de ce module (on n'aura peut-être pas le temps de regarder les symétries...).