Cours n°8: Variantes d'algorithmes de recherche arborescente pour l'optimisation combinatoire (et l'intelligence artificielle)

Nicolas DUPIN

`https://github.com/ndupin/ORteaching`
`http://nicolasdupin2000.wixsite.com/research`

version du 20 mars 2022

Issu et étendu d'enseignements donnés à l'ENSTA Paris, Polytech' Paris-Saclay, et à l'université Paris-Saclay

Rappel : algorithme Branch&Bound (B&B) en PLNE

Pour un problème de minimisation, les principes de l'algorithme de Branch&Bound (ou séparation-évaluation) guidés par la relaxation PL :

- La résolution continue, résolution PL, donne des bornes inférieures.
- élagage : un noeud est élagué (inutile à explorer) si sa borne inférieure est supérieure à (au moins aussi bonne que) la meilleure solution connue.
- Branchements : restreindre les valeurs possibles d'une variable, définit un parcours arborescent (arbre binaire) ,
- Branchements et bornes : ajouter des contraintes linéaires, préserve la structure PLNE, on peut utiliser les bornes PL à chaque noeud. Cela augmente la borne inférieure.

$\Longrightarrow$ Algorithme B&B convergeant vers une solution optimale, ou en temps arrêté fournit majoration et minoration de la valeur optimale.

Rappels : Eléments clés pour l'efficacité pratique B&B

Avoir plus rapidement une bonne solution (primale) peut permettre de supprimer des noeuds plus tôt :

Améliorer les bornes inférieures à un noeud, peut permettre de supprimer des noeuds plus tôt :

- Le travail de modélisation PLNE influe sur la qualité des relaxations continues.
- Ajout de coupes d'intégrité : contraintes valides sur les points entiers permettant de couper des solutions fractionnaires.

Avoir plus rapidement une bonne solution (primale) peut permettre de supprimer des noeuds plus tôt :

- Heuristiques génériques pour chercher des bonnes solutions à chaque noeud.
- Démarrage à chaud : fournir initialement une bonne solution (primale) accélère la résolution PLNE. Une heuristique spécialisée en amont de la PLNE par exemple.

$\Longrightarrow$ Compromis à trouver entre l'allongement des temps de calculs à chaque noeud, et le temps gagné à énumérer moins de noeuds.

Plan

Plan

Rappel : Approches polyédrales

- On peut parfois avoir une description avec un nb exponentiel de contraintes de l'enveloppe convexe des points entiers.
- Démonstration : prouver que des contraintes définissent des facettes de l'enveloppe convexe.
- Dans certains cas , on peut résoudre un tel PL avec un nb exponentiel de contraintes en temps polynomial (on le verra au prochain cours).
- Autre outil pour prouver que des problèmes d'optimisation (ou des sous-cas particuliers) sont polynomiaux

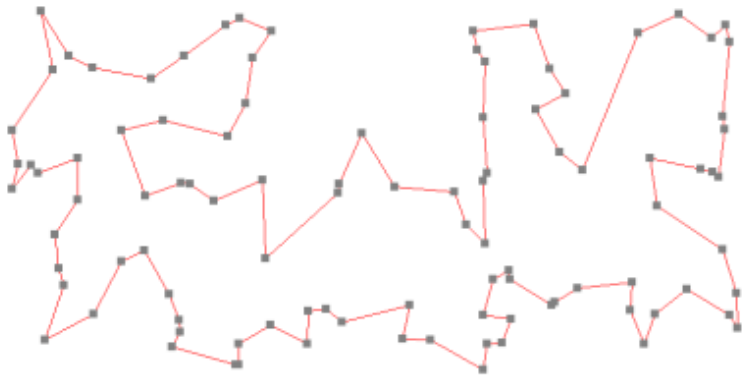
$\Longrightarrow$ Comment résoudre un PL avec un nb exponentiel de contraintes ?

`https://www.lamsade.dauphine.fr/~mahjoub/MODO/Chapitre-Polyedres.pdf`

`http://ejc-gdr-ro.event.univ-lorraine.fr/docs/RM.pdf`

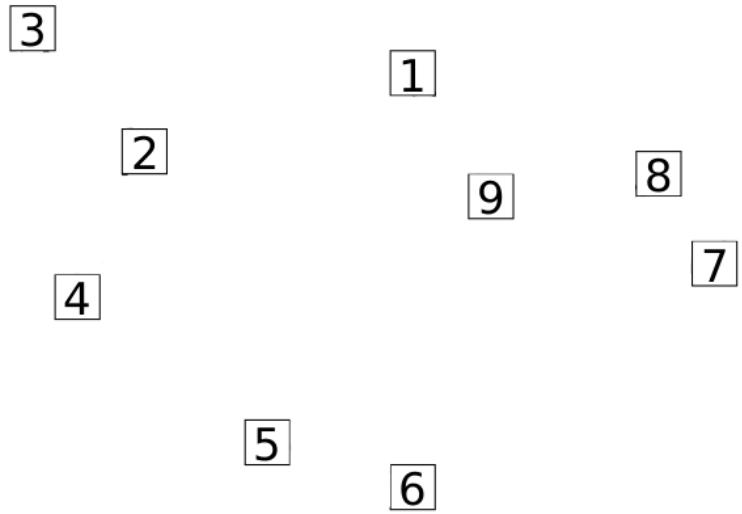
Problèmes de voyageur de commerce, notations

- Un voyageur de commerce doit transiter par N villes, $\mathcal{N} = [\![1; N]\!]$.
- Tous les trajets sont possibles, la distance entre les villes $n \in \mathcal{N}$ et $n' \in \mathcal{N}$ est $d_{n,n'} = d_{n',n}$ (sinon, $d_{n,n'} = +\infty$ ou un majorant).
- Voyageur de commerce symétrique (TSP) : $d_{n,n'} = d_{n',n}$, graphe non orienté.
- Voyageur de commerce asymétrique (ATSP) : on n'a pas forcément $d_{n,n'} = d_{n',n}$, structure de graphe orienté.
- Minimiser la longueur du trajet du représentant de commerce.



$\Longrightarrow$ Un problème d'optimisation fortement NP-complet

Modélisation PLNE du pb de voyageur de commerce



On note :

- $\mathcal{N}$ l'ensemble des villes et $d_{n,n'}$ la distance entre les villes n et n'.
- Variables binaires $x_{n,n'} \in \{0,1\}$ pour $(n,n') \in \mathcal{N}^2$ avec $n' \neq n$.
- $x_{n,n'} = 1$ ssi le voyageur de commerce va dans la ville n' immédiatement après avoir visité la ville n.
- Objectif à minimiser $\sum_{n,n'} d_{n,n'} x_{n,n'}$.

Formulation PLNE du TSP avec contraintes de sous-tours

$$
\begin{aligned}
OPT = \min_{x} \quad & \sum_{n' \neq n} d_{n,n'} x_{n,n'} \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n,n'} = 1 \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n',n} = 1 \\
\forall \mathcal{S} \subset \mathcal{N}, \quad & \sum_{n \neq n' \in \mathcal{S}} x_{n,n'} + x_{n',n} \leqslant |\mathcal{S}| - 1 \\
\forall n' \neq n \quad & x_{n,n'} \in \{0,1\},
\end{aligned}
\tag{1}
$$

$x_{n,n'} = 1$ ssi le voyageur de commerce va dans la ville n' immédiatement après avoir visité la ville n.

Nombre non polynomial de contraintes, contraintes de sous-tour indexées sur les parties d'un ensemble à N éléments

$\Longrightarrow$ Comment résoudre une telle formulation PLNE ?

PLNE avec un sous ensemble de sous-tours

Soit $\mathcal{S}' \subset \mathcal{P}(\mathcal{N})$ un ensemble de sous ensemble de villes. On considère le PLNE restreint interdisant uniquement les sous-tours de $\mathcal{S}'$:

$$
\begin{aligned}
LB(\mathcal{S}') = \min_{x} \quad & \sum_{n' \neq n} d_{n,n'} x_{n,n'} \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n,n'} = 1 \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n',n} = 1 \\
\forall \mathcal{S} \in \mathcal{S}', \quad & \sum_{n \neq n' \in \mathcal{S}} x_{n,n'} + x_{n',n} \leqslant |\mathcal{S}| - 1 \\
\forall n' \neq n \quad & x_{n,n'} \in \{0,1\},
\end{aligned}
\tag{2}
$$

$LB(\mathcal{S}') \leqslant OPT$ car problème moins contraint que le problème original.

Si la (ou une) solution optimale retournée pour $LB(\mathcal{S}')$ vérifient toutes les contraintes de sous-tour, même les non écrites, alors cette solution est optimale dans le pb original et $LB(\mathcal{S}') = OPT$

$\Longrightarrow$ Comment vérifier que la solution fournie pour $LB(\mathcal{S}')$ vérifie les 2^N contraintes de sous tour ?

Rappel : Relations entre bornes

Soit A, B dans $\mathbb{R}^n$ avec $A \subset B$, soit $f : B \longrightarrow \mathbb{R}$ une fonction bornée.

$\inf_{x \in A} f(x) = \inf\{f(x) | x \in A\}$, $\inf_{x \in B} f(x)$, $\sup_{x \in A} f(x) = \sup\{f(x) | x \in A\}$, $\sup_{x \in B} f(x)$ sont bien définis et :

$$\inf_{x \in A} f(x) \geqslant \inf_{x \in B} f(x) \tag{3}$$

$$\sup_{x \in A} f(x) \leqslant \sup_{x \in B} f(x) \tag{4}$$

(plus on a d'éléments à considérer, plus le minimum est bas)
(plus on a d'éléments à considérer, plus le maximum est élevé)

On aura appliqué ce résultat pour la relaxation continue, par l'ajout de contraintes et de coupes.

Dans le cas où B est bornée dans $\mathbb{R}^n$, f est continue, les hypothèses sont vérifiées. Si de plus B est fermé (alors compact), les bornes inférieures et supérieures sont des min et max : les bornes sont atteintes.

Séparation entière pour le TSP

Question de séparation entière : Comment vérifier que la solution fournie pour $LB(\mathcal{S}')$ vérifie les 2^N contraintes de sous-tour ? Comment fournir un sous-tour sans énumérer les $2^N - |\mathcal{S}'|$ possibilités ?

On a la donnée de $x_{n,n'} \in \{0,1\}$ avec $x_{n,n'} = 1$ ssi le voyageur de commerce va dans la ville n' immédiatement après avoir visité la ville n.

Etape 0 : On part d'une ville, par exemple la ville 1, que l'on marque.

Etape 1 : Soit n la ville courante. On calcule n' la ville suivant la ville courante, ie telle que $x_{n,n'} = 1$.

Etape 2 : Si $n' \neq 1$ la ville initiale, on marque la ville n' et on revient à l'étape 1 avec $n = n'$.

Etape 3 : On a $n' = 1$ la ville initiale. Si tous les sommets ont été marqués, la solution x ne continent pas de sous tour, sinon, les sommets marqués forment un sous-tour dans la solution x

$\Longrightarrow$ Algorithme en $O(N^2)$ pour répondre à la question de séparation entière, et fournir un sous tour

Séparation entière pour le TSP : exemple (1)

$x_{i,j}$	$j=$ 1	2	3	4	5
$i=1$	0	1	0	0	0
$i=2$	0	0	0	1	0
$i=3$	0	0	0	0	1
$i=4$	1	0	0	0	0
$i=5$	0	0	1	0	0

$x_{1,2} = 1$, $x_{2,4} = 1$, $x_{4,1} = 1$

On a dans x le sous-tour :
$1 \to 2 \to 4 \to 1$

$x_{3,5} = 1$, $x_{5,3} = 1$

On a dans x le sous-tour $3 \to 5 \to 3$

2 Coupes à ajouter :

- Pour le sous-tour : $1 \to 2 \to 4 \to 1$

$x_{1,2} + x_{1,4} + x_{2,4} + x_{4,2} + x_{4,1} + x_{2,1} \leqslant 2$

- Pour le sous-tour $3 \to 5 \to 3$:

$x_{3,5} + x_{5,3} \leqslant 1$

N.B : ces deux expressions linéaires valaient 3 et 2 dans la solution partielle x

Séparation entière pour le TSP : exemple (2)

$x_{i,j}$	$j=$ 1	2	3	4	5	6	7
$i=1$	0	1	0	0	0	0	0
$i=2$	0	0	0	1	0	0	0
$i=3$	0	0	0	0	0	1	0
$i=4$	1	0	0	0	0	0	0
$i=5$	0	0	0	0	0	0	1
$i=6$	0	0	0	0	1	0	0
$i=7$	0	0	1	0	0	0	0

$x_{1,2} = 1$, $x_{2,4} = 1$, $x_{4,1} = 1$

On a dans x le sous-tour :
$1 \to 2 \to 4 \to 1$

$x_{3,6} = x_{6,5} = x_{5,7} = x_{7,3} = 1$

On a dans x le sous-tour :
$3 \to 6 \to 5 \to 7 \to 3$

2 Coupes à ajouter :

- Pour le sous-tour : $1 \to 2 \to 4 \to 1$

$x_{1,2} + x_{1,4} + x_{2,4} + x_{4,2} + x_{4,1} + x_{2,1} \leqslant 2$

- Pour le sous-tour $3 \to 6 \to 5 \to 7 \to 3$:

$x_{3,5} + x_{5,3} + x_{3,6} + x_{6,3} + x_{3,7} + x_{7,3} + x_{5,6} + x_{6,5} + x_{5,7} + x_{7,5} + x_{7,6} + x_{6,7} \leqslant 3$

N.B : ces deux expressions linéaires valaient 3 et 4 dans la solution partielle x

Algorithme de séparation entière pour le TSP

Algorithme (pseudo-code) : Séparation entière pour le TSP

Entrée : • Un problème de TSP défini par N villes et les distances $d_{n,n'} \geqslant 0$
• $x_{n,n'} \in \{0,1\}$ une solution entière avec sous tour potentiels
Sortie : SEPARATIONENTIERETSP(villeInit) renvoie s'il existe un sous tour
N.B :fournit le sous tour de la ville villeInit

SEPARATIONENTIERETSP(villeInit, TSP, $x_{n,n'}$)
initialise compt := 1, villeCourante := villeInit
initialise l un vector de n booléens à false sauf l[villeInit] :=true
while $x[villeCourante, villeInit] \neq 1$:
find v such that $x[villeCourante, v] = 1$
l[v] :=true ; villeCourante :=v ; compt := compt+1
end while
if compt == N **then return** False, $\emptyset$
else return True, liste des villes v tq l[v]==true
end if

Algorithme (pseudo-code) : Séparation entière pour le TSP

Entrée : • Un problème de TSP défini par N villes et les distances $d_{n,n'} \geqslant 0$
• $x_{n,n'} \in \{0,1\}$ une solution entière avec sous tour potentiels
Sortie : SEPARATIONENTIERETSP(villeInit) renvoie tous les sous tours

SEPARATIONENTIERETSP TSP, $x_{n,n'}$)
initialise compt := 1, villeInit := 1, villeCourante := 1
initialise marked un vector de n booléens à false sauf marked[villeInit] :=True
initialise subTours à vide un sous ensemble de $\mathcal{P}(\mathbb{N})$, current :=$\emptyset$
while compt< N :
find v such that $x[villeCourante, v] = 1$
if v == villeInit **then** :
add current in subTours ; clear current
villeInit := villeCourante := the smallest v such that marked[v] :=False
else villeCourante :=v ; add v in current
end if
marked[villeCourante] :=True ; compt := compt+1
end while
if subTours.size() == 1 **then return** False, $\emptyset$
else return True, subTours
end if

Algorithme par itérations séquentielles de PLNE

Algorithme séquentiel, rudimentaire, mais déjà assez efficace en général :

- Etape 1 : on sélectionne un ensemble initial $\mathcal{S}'$ de contraintes de sous-tour (aucune ou sous tours entre deux villes).
- Etape 2 : On résout (à l'optimalité) le PLNE avec les contraintes de sous-tour de $\mathcal{S}'$? Soit x une solution et $LB(\mathcal{S}')$ la valeur du PLNE.
- Etape 3 : On appelle la séparation entière sur x. Si on a trouvé un sous-tour dans x (ou qu'on en a calculé plusieurs), on l' (ou les) ajoute dans $\mathcal{S}'$ et on revient à l'étape 2
- Etape 4 : On retourne la solution x alors optimale et $LB(\mathcal{S}') = OPT$

$\Longrightarrow$ Quelle implémentation, quelles faiblesses et comment améliorer cet algorithme naïf ?

N.B : un tel algorithme est codé fourni dans les exemples d'OPL et Cplex.

Algorithme séquentiel, analyse et implémentation

- A chaque itération, $LB(\mathcal{S}')$ est un minorant de l'optimum, cette borne est croissante.
- Succession de calculs de PLNE, donc de parcours arborescents repartant de zéro : la solution précédente n'est pas réalisable, inutilisable directement pour un démarrage à chaud par exemple.
- Avec une heuristique en amont, calculant une solution primale du TSP, cette solution fait un démarrage à chaud pour tous les calculs de PLNE, pour les accélérer.
- Amélioration de borne primale : on peut essayer de trouver une solution réalisable à partir de la solution courante x avec sous-tours, réparation en joignant des villes appartenant à des sous-tours

$\Longrightarrow$ Le point critique, itérer des parcours arborescents potentiellement exponentiels, avec des potentielles redondances.

$\Longrightarrow$ Peut on converger uniquement en utilisant un seul parcours arborescent ?

Implémentation Branch & Cut avec séparation entière

- But : parcourir un seul arbre obtenu par branchements sur les variables x, pas de différence avec Branch&Bound.
- Une heuristique peut initialiser avec une solution primale.
- Une borne de relaxation continue avec un nombre restreint de coupes de sous-tour et des contraintes de branchements est un minorant de la borne associée au noeud en considérant toutes les contraintes : c'est une borne valide.
- Critère d'élagage est toujours valide, entre une borne continue d'un noeud et la meilleure solution réalisable sans sous-tour
- Différence avec Branch&Bound : quand une solution entière est trouvée par calcul de relaxation continue, il faut appeler la séparation entière au noeud. En cas de sous tour, on rappelle la résolution PL au noeud avec la coupe de sous-tour à rajouter. On actualise la meilleure solution courante après avoir vérifié la validité (ie pas de sous tour)

Et si les branchements forment un sous -tour ?

- La séparation entière va le détecter sur les solutions entières générées.
- Dans ce cas, on a un noeud infaisable sur la relaxation continue après ajout de la coupe de sous tour.
- Une telle difficulté peut être détectée trop tardivement.
- On peut renforcer les branchements : avec des fixations partielles, ça impose de retirer les variables qui formeraient des sous-tours.
- Ici, quand on impose une variables $x_{n,n'}$ à 1, on regarde les composantes connexes de n et n', et on retire les variables reliant chaque coupe de variable entre les deux composantes connexes (si les componsates connexes ne contiennent pas toutes les villes).

Relaxation continue du TSP avec contraintes de sous-tours

Question, peut on calculer la relaxation continue, PL avec un nombre exponentiel de contraintes :

$$
\begin{aligned}
LP_{OPT} = \min_{x} \quad & \sum_{n' \neq n} d_{n,n'} x_{n,n'} \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n,n'} = 1 \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n',n} = 1 \\
\forall \mathcal{S} \subset \mathcal{N}, \quad & \sum_{n \neq n' \in \mathcal{S}} x_{n,n'} + x_{n',n} \leqslant |\mathcal{S}| - 1 \\
\forall n' \neq n \quad & x_{n,n'} \geqslant 0,
\end{aligned}
\tag{5}
$$

On avait mentionné que de telles descriptions polyédrales sont meilleures que les formulations compactes type MTZ, GP, SSB, FFG, . . ., et donc donnent de meilleures relaxations continues

$\Longrightarrow$ Comment résoudre de tels PL ? Comment en tirer partie dans un algorithme de Branch&Cut de résolution des solutions entières du TSP ?

PLNE avec un sous ensemble de sous-tours

Soit $\mathcal{S}' \subset \mathcal{P}(\mathcal{N})$ un ensemble de sous ensemble de villes. On considère le PL restreint interdisant uniquement les sous-tours de $\mathcal{S}'$:

$$
\begin{aligned}
LP(\mathcal{S}') = \min_{x} \quad & \sum_{n' \neq n} d_{n,n'} x_{n,n'} \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n,n'} = 1 \\
\forall n \in \mathcal{N}, \quad & \sum_{n' \neq n} x_{n',n} = 1 \\
\forall \mathcal{S} \in \mathcal{S}', \quad & \sum_{n \neq n' \in \mathcal{S}} x_{n,n'} + x_{n',n} \leqslant |\mathcal{S}| - 1 \\
\forall n' \neq n \quad & x_{n,n'} \geqslant 0,
\end{aligned}
\tag{6}
$$

$LP(\mathcal{S}') \leqslant LP_{OPT}$ car problème moins contraint que le problème original.

Si la (ou une) solution optimale (continue ici) retournée pour $LP(\mathcal{S}')$ vérifie toutes les contraintes de sous-tour, même les non écrites, alors cette solution est optimale dans le pb original et $LP(\mathcal{S}') = LP_{OPT}$

$\Longrightarrow$ Comment vérifier que la solution continue fournie pour $LP(\mathcal{S}')$ vérifie les 2^N contraintes de sous tour ?

Séparation continue pour le TSP (1)

Question de séparation continue : Comment vérifier que la solution fournie pour $LP(\mathcal{S}')$ vérifie les 2^N contraintes de sous-tour ? Comment fournir un sous-tour sans énumérer les $2^N - |\mathcal{S}'|$ possibilités ?

On a la donnée de $\tilde{x}_{n,n'} \in [0,1]$, valeurs continues.

Moins direct et moins facile à vérifier que pour la séparation entière

Soit $G = (V, E)$ le graphe orienté pondéré avec $V = \mathcal{N}$, $e = (n, n') \in E$ désigne un couple de noeuds tel que $\tilde{x}_{n,n'} > 0$ et de poids $w_e = \tilde{x}_{n,n'} > 0$

Si un sous-tour existe, on a deux parties du graphes isolées, ie une coupe entre deux points inférieure à 1 dans le graphe pondéré orienté

Vérification : pour tout $n \neq 1$, on vérifie que la coupe minimale (dual de max-flot) dans le graphe pondéré G entre 1 et n est de valeur au moins 1.

Si on a une coupe minimale entre un $n \neq 1$ et 1 de valeur < 1, la partition en deux parties du graphe fournit deux coupes de sous-tour.

N.B : avec formulation symétrique du TSP, version orientée du graphe pondéré et coupes de valeur 2.

Séparation continue pour le TSP (2)

On a la donnée de $\tilde{x}_{n,n'} \in [0,1]$, valeurs continues.

Soit $G = (V, E)$ le graphe orienté pondéré avec $V = \mathcal{N}$, $e = (n, n') \in E$ désigne un couple de noeuds tel que $\tilde{x}_{n,n'} > 0$ et de poids $w_e = \tilde{x}_{n,n'} > 0$

On vérifie pour tout $n \neq 1$, que la coupe minimale (algo de Gomory- Hu, Padberg- Rinaldi, Hao -Orlint . . .plutôt que calcul PL) dans le graphe pondéré G entre 1 et n est de valeur au moins 1.

Si une coupe minimale de valeur < 1 est trouvée, pas besoin de faire les autres calculs, on retourne deux coupes de sous-tour induites par la partition en deux parties du graphe fournit.

Sinon, on a fait $N - 1$ calculs de coupe minimale, complexité polynomiale, pour prouver que $\tilde{x}$ satisfait toutes les contraintes de sous tour.

Algorithme par itérations séquentielles de PL

Algorithme similaire à la résolution séquentielle précédente, sauf que l'on n'a que des PL :

- Etape 1 : on sélectionne un ensemble initial $\mathcal{S}'$ de contraintes de sous-tour (aucune ou sous tours entre deux villes).
- Etape 2 : On résout le PL avec les contraintes de sous-tour de $\mathcal{S}'$. Soit $\tilde{x}$ une solution et $LP(\mathcal{S}')$ la valeur du PLNE.
- Etape 3 : On appelle la séparation continue sur $\tilde{x}$. Si on a trouvé des sous-tour dans $\tilde{x}$, on les)ajoute dans $\mathcal{S}'$ et on revient à l'étape 2
- Etape 4 : On retourne la solution $\tilde{x}$ et $LP(\mathcal{S}') = LP_{OPT}$

Ici, contrairement aux PLNE séquentiels, la solution $\tilde{x}$ de l'itération précédente permet d'initialiser le simplexe dual de l'itération suivante. Les itérations successives de PL sont alors bien plus rapides, élément clé en pratique.

$\Longrightarrow$ Avec simplexe dual pour itérations successives, un bon schéma pour résoudre efficacement un tel PL avec un nombre exponentiel de contraintes.

Théorème de Lovasz-Schrijver-Grötschel

Définition (Algorithme de séparation)

L'algorithme de séparation est l'algorithme de choix de contrainte à ajouter à une formulation restreinte en contraintes et à sa solution optimale partielle : soit assurer que la solution partielle est optimale du problème complet, soit fournir une contrainte non satisfaite.

Théorème fondamental de l'analyse polyédrale :

Théorème (Théorème de Lovasz-Schrijver-Grötschel)

Pour un Programme Linéaire, si il existe un algorithme de séparation polynomial, alors le calcul du PL est aussi polynomial.

L'algorithme précédent d'ajout dynamique des contraintes converge alors en un nombre polynomial d'itérations.

Grötschel, M., Lovász, L., & Schrijver, A. (1981). The ellipsoid method and its consequences in combinatorial optimization. Combinatorica, 1(2), 169-197.

Théorème de Lovasz-Schrijver-Grötschel, discussions

- Cas PL, avec nombre polynomial de contraintes : le théorème de Lovasz-Schrijver-Grötschel est plus fort que la résolution d'un PL est polynomiale.
- Approche polyédrale, deux arguments pour montrer qu'un PLNE est polynomial : avoir une description linéaire des facettes du polyèdre de l'enveloppe convexe des points entiers, et avoir une séparation polynomiale
- Cas TSP : séparation polynomiale, mais la description linéaire avec les contraintes de sous-tour ne décrit pas l'enveloppe convexe des points entiers.
- Importance de connaître ses classiques, les problèmes polynomiaux et NP-complets pour al séparation
- Approche Branch&Cut pour des pb NP-complets : ce sont des approches avec une complexité de pire cas exponentielle, mais assez efficaces en pratique. On se soucie d'avoir une séparation efficace, et pas que polynomiale. Une heuristique rapide peut replacer une séparation exacte tant que l'on trouve des coupes à ajouter.

Algorithme Branch & Cut pour le TSP (1)

- Une heuristique peut initialiser avec une solution primale.
- A chaque noeud du parcours arborescent, on ajoute les coupes continues de sous tour nécessaires au calcul à l'optimalité du PL avec toutes le contraintes de sous tour. Le calcul continu s'arrête quand aucune coupe ne peut être ajoutée. (on peut aussi avoir des coupes d'intégrité)
- Les bornes à chaque noeud sont meilleures qu'avec la séparation entière.
- Quand une solution entière est trouvée par convergence de relaxation continue avec coupes, elle est nécessairement réalisable.
- Critère d'élagage est toujours valide, entre une borne continue d'un noeud et la meilleure solution réalisable sans sous-tour

Algorithme Branch&Cut pour le TSP (2)

- On peut hybrider les algorithmes de Branch& Cut avec séparations entières et continues.
- Comme les coupes d'intégrité, les coupes de sous-tour en trop grand nombre peuvent alourdir les calculs
- On peut alors avoir une génération de coupes uniquement quand cela améliore significativement la relaxation continue. Dans ce cas, il faut appliquer la séparation entière sur les solutions entières trouvées avec une génération partielle de coupes de sous-tour.
- N.B : un tel critère d'arrêt existe pour les coupes d'intégrité.
- Mode hybride utilisé souvent par les solveurs, ne garantissent pas que toutes les séparations sont appelées.

Coupes locales ou globales ?

- Générer bcp de coupes peut alourdir les itérations de type simplexe dual pour les calculs PL.
- Question empirique : une coupe ajoutée à un noeud doit elle être ajoutée à tous les noeuds (coupes globales), uniquement au noeud courant, ou au noeud courant et à ses enfants (coupes locale) ?
- Un paramètre empirique supplémentaire !
- Les séparations entières sont appelées peu souvent et peuvent concerner bcp de noeuds : peuvent être globales.
- Les séparations continues à un noeuds peuvent être nombreuses, on peut ne garder pour les noeuds enfants que les contraintes saturées sur la dernière itération, et effacer des coupes intermédiaires dans le noeud qui ne sont plus saturées.
- Gestion intermédiaire : avoir un pool de coupes déjà générées, à re-parcourir rapidement pour vérifier si on a des coupes à rajouter parmi le pool.
- N.B : certaines questions se posent aussi pour les coupes d'intégrité.

Les callbacks : pour interagir avec un solveur Branch&Cut

Les callbacks (non disponibles avec les modeleurs), disponibles avec les API des principaux solveurs de PLNE permettent d'implémenter des composants de Branch&Cut spécifiques :

- Récupérer l'état d'un noeud courant, et générer une heuristique primale pour accélérer la résolution PLNE.
- Récupérer les solutions réalisables entières trouvée pour leur appliquer une séparation entière, ajout de coupe.
- Générer des coupes (intégrité un de formulation) à partir d'une solutions continue.

Avec Julia JuMP (pour les principaux solveurs avec ces fonctionalités) et pour Python-MIP (pour CBC et Gurobi), on a de tels callbacks avec une interface générique à plusieurs solveurs. Python-MIP fournit même comme exemple le TSP avec séparation continue :

`https://python-mip.readthedocs.io/en/latest/custom.html`

Un peu de généralité sur les algorithmes de Branch&Cut

- Le TSP permettait de montrer les principes des algorithmes de Branch&Cut avec des exemples de séparation.
- L'algorithme de Branch&Cut s'utilise généralement pour de nombreux problèmes (dont des pb d'optimisation dans les graphes coloration)
- Opérateur clé : la séparation entière et/ou continue, à déterminer au cas par cas et à implémenter finement.
- Principes généraux une fois que la séparation est définie, les implémentations sont également génériques avec les solveurs, seule la séparation est à définir.

Pour aller plus loin sur le TSP

Sur le TSP (et l'ATSP), il existe de multiples versions d'algorithmes de Branch&Cut, différentes formulations et différents séparations induites :

`http://www.dmf.unicatt.it/iniziative/scuola_geom/2004/Note/Lodi/Lodi_Lecture_1.pdf`

Le solveur exact de l'état de l'art, CONCORDE, est un algorithme de Branch&Cut

`https://www.math.uwaterloo.ca/tsp/concorde.html`

Plan

Rappels

Pour présenter l'algorithme de séparation et évaluation général, deux exemple illustratifs

- Problème du Voyageur de commerce (TSP) à n villes : les distances entre deux villes sont connues, minimiser la distance du cycle qui passe par toutes les villes.

- Encodage du TSP : ville 1 point de départ et d'arrivée, et une permutation de $[\![2, n]\!]$

- Problème de stable max (StabMax) : étant donné un graphe $G = (V, E)$ on veut sélectionner un sous ensemble de sommets de cardinal maximal, tq tous les sommets sélectionnés ne sont pas reliés entre eux 2 à 2 dans G.

- Encodage de StabMax : un vecteur/tableau de $\{0, 1\}^{|V|}$, on indique pour chaque sommet si est sélectionné dans le stable (valeur 1) ou non sélectionné (valeur 0).

Algorithmes de recherche exhaustive “brute force”

- La PLNE définit dans le cas borné (ex que des variables binaires) un ensemble fini de solutions entières.
- L'algorithme "brute force" est une recherche exhaustive : évaluer toutes les solutions possibles et déterminer celle qui est optimale.
- TSP à n villes : $(n-1)!$ possibilités à énumérer ...
- StabMax : $2^{|V|}$ possibilités en énumérant tous les sous ensembles de $|V|$, vérifier que l'ensemble est stable et au besoin actualiser la meilleure valeur trouvée pour le cardinal d'un stable ...

$\Longrightarrow$ Algorithmes brute force impraticable même pour des valeurs de n assez petites ($20! = 2.432.902.008.176.640.000$)

Algorithmes de séparation évaluation “Branch&Bound”

- Dans la pratique, on essaie d'énumérer moins de solutions afin de réduire le temps de calcul de l'algorithme exact :
- L'algorithme de Branch&Bound (séparation-évaluation en français) : utilise une borne "optimiste" pour évaluer des énumérations partielles (bornes inférieures pour un problème de minimisation, supérieures pour une maximisation)
- On a un schéma d'énumération totale de toutes les solutions.
- Si pour une énumération partielle donnée, la borne optimiste est moins bonne que la meilleure solution trouvée, rien ne sert d'énumérer des sous-cas, pas de solution optimale contenant l'énumération partielle.

N.B : on parle de bornes “duales” pour ces bornes optimistes, les bornes "primales" désignant les bornes définies par des scores de solutions réalisables. (pour éviter les confusions borne inf/sup selon que l'on considère une maximisation ou une minimisation)

Algorithmes force brute : cas TSP

- Comment énumérer toutes les sous ensembles de $[\![2, n]\!]$?
- Ex : énumérer dans un ordre lexicographique, de $[2; 3; \ldots; n]$ à $[n; n-1; \ldots; 2]$ ou de $[n; n-1; \ldots; 2]$ à $[2; 3; \ldots; n]$.
- Programmation récursive : à partir d'une sous-liste des premiers sommets visités, énumérer toutes les possibilités pour la ville suivante, non incluses dans la liste des villes visitées, et appeler récursivement pour les différents ajouts. Une fois toutes les villes visitées, on calcule le coût, et on compare à la meilleure solution courante.

Algorithmes force brute : cas TSP

Algorithme (pseudo-code) : énumération des solutions pour le TSP

Entrée : • Un problème de TSP défini par N villes et les distances $d_{n,n'} \geqslant 0$
• $x_{n,n'} \in \{0,1\}$ une solution entière avec sous tour potentiels
Sortie : ENUMTSP(0, 1, $\{1\}$) affiche toutes les solutions du TSP et leur cout

```
ENUMTSP(cost, last, list)
  if liste.size()==N : print(cost+d_{last,1}, list) and return
  for all n ∈ 𝒩-list :
    list = n : :list
    ENUMTSP(cost+d_{last,n}, n, list)
    list = tl(list) // remove n
  end for
```

Algorithme séparation&évaluation : cas TSP

- On stocke la meilleure solution trouvée, peut être initialisée avec une bonne heuristique (ex : glouton, GRASP)
- Schéma d'énumération : schéma d'énumération partielle selon les premières villes visitées (comme précédemment)
- Borne optimiste 1 : distance totale minorée par les distances définies par les premières villes. (car les distances sont positives, minoration par 0.)
- Borne optimiste 2 : Borne optimiste 1 + distance de la première ville à la dernière ville de l'énumération partielle. (marche avec inégalité triangulaire, distances Euclidiennes)
- Améliorer avec un minorant d'un chemin élémentaire de ville courante à la ville origine ? (PL peut faire l'affaire, ou autres minorations)
- Dans une énumération partielle $[a; b; \ldots, m]$, on stocke la distance parcourue jusqu'à m. Si une énumération partielle a une borne plus grande que la meilleure solution courante, on ne va pas plus loin.

Algorithme séparation&évaluation : cas TSP

Algorithme (pseudo-code) : séparation&évaluation pour le TSP

Entrée : • Un problème de TSP défini par N villes et les distances $d_{n,n'} \geqslant 0$
• $x_{n,n'} \in \{0,1\}$ une solution entière avec sous tour potentiels
Variables globales : bestCost, bestSolution, list
Sortie : BBTSP(0, 1) avec list $= \{1\}$ calcule une solution optimale du TSP

```
BBTSP(cost, last)
  if list.size()==N :
    if cost+d_{last,1} <bestCost : update bestCost, bestSolution end if
    return
  end if
  for all n ∈ N-list :
    list = n : :list
    if cost+d_{last,n}+LB(N-list)⩾bestCost : BBTSP(cost+d_{last,n}, n) end if
    list = tl(list)
  end for
```

Algorithmes force brute : cas maxStable

- Enumérer toutes les sous ensembles de $|V|$: assez facile.
- Ex : dans l'ordre lexicographique, de $[0; 0; \ldots; 0]$ à $[1; 1; \ldots; 1]$, possibilités à énumérer peuvent être encodées sur un arbre binaire.
- Mieux : essayer d'énumérer uniquement les stables dans l'ordre lexicographique inverse.
- On part de $[1; x; x; \ldots]$. chercher le stable maximal dans l'ordre lexicographique revient à imposer 1 comme première valeur, imposer des 0 aux voisins du noeud choisi, et itérer tant que des sommets sont non affectés en choisissant le premier sommet de l'ordre lexicographique.
- Algorithme de backtrack : une fois un stable trouvé, on revient en arrière sur la (ou les) dernière(s) décision(s) de fixation à 1, pour parcourir les stables sur l'ordre lexicographique inverse
- Un tel parcours se code aisément avec de la récursivité

Algorithme séparation & évaluation : cas maxStable

- On stocke la meilleure solution trouvée (peut être initialisée avec une bonne heuristique (ex : glouton, GRASP)
- Schéma d'énumération : schéma d'énumération partielle des stables
- Borne optimiste : au plus, tous les sommets non fixés à 0 ou 1 sont des 1.
- Borne optimiste : cardinal des fixés à 1 + cardinal des sommets non reliés avec les sommets fixés
- Si une énumération partielle $[1; 0; 1; x; x; 0; x; \dots]$ a une borne inférieure à la meilleure solution courante, on appelle le backtrack à ce stade.

Amélioration de borne optimiste : cas maxStable

- On reprend la stratégie précédente.
- Parmi les sommets non fixés à 0 ou 1, avoir deux sommets reliés dans G induit que l'on ne peut avoir les deux simultanément dans un stable : la borne optimiste baisse de 1.
- Algo récursif : on considère initialement V' les noeuds non fixés, et $G' = (V', E')$ la restriction de G à V'. Soit $B = |V'|$
- Etape 1 : Si $|E'| = 0$, V' est un stable, $B = |V'|$ est une borne exacte que l'on renvoie. Sinon, étape 2
- Etape 2 : On sélectionne une arête (v_1, v_2). Soit $V'' = V - \{v_1, v_2\}$, et E'' les arrêtes de E' qui ne contiennent pas v_1, v_2.
- Etape 3 : Une borne optimiste : $1 +$ borne(V'', E''), on revient à l'étape 1 avec $V'' = V'$ et $E'' = E'$.

$\Longrightarrow$ Dans le meilleur des cas, on a divisé la borne approximée par 2

Amélioration de borne avec des cliques

Soit V_1, V_2 un partition de V, soient E_1, E_2 les arêtes de E parmi deux sommets de V_1 et V_2 respectivement

$\text{maxStable}(V, E) \leqslant \text{maxStable}(V_1, E_1) + \text{maxStable}(V_2, E_2)$

N.B : se démontre directement en relâchant dans le PLNE les contraintes sur les arêtes entre V_1 et V_2.

Sur une clique (V_1, E_1) : $\text{maxStable}(V_1, E_1) = 1$

On peut partitionner les sommets de V selon des cliques (algos gloutons pour cela).

Le nombre de cliques recouvrant les sommets est une borne supérieure

Facile à calculer, et améliore nettement les bornes naïves précédentes

N.B : on peut calculer plusieurs recouvrement par des cliques de manière gloutonnes, pour prendre la plus petite majoration

Bornes par recouvrement de cliques, cas pondéré

On considère le cas pondéré, avec des poids c_v pour chaque somme $v \in V$

Soit V_1, V_2 un partition de V, soient E_1, E_2 les arêtes de E parmi deux sommets de V_1 et V_2 respectivement

$\text{WeightMIS}(V, E) \leqslant \text{WeightMIS}(V_1, E_1) + \text{WeightMIS}(V_2, E_2)$

N.B : se démontre directement en relâchant dans le PLNE les contraintes sur les arêtes entre V_1 et V_2.

Sur une clique (V_1, E_1) : $\text{maxStable}(V_1, E_1) = \max_{v \in V_1} c_v$

On peut partitionner les sommets de V selon des cliques (algos gloutons pour cela).

On peut alors calculer facilement un majorant par recouvrement des sommets selon des cliques.

N.B : on peut calculer plusieurs recouvrement par des cliques de manière gloutonnes, pour prendre la plus petite majoration

Références : Branch&Bound pour cliqueMax

Tomita, E., & Kameda, T. (2007). An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. Journal of Global optimization, 37(1), 95-111.

Tomita, E., & Seki, T. (2003). An efficient branch-and-bound algorithm for finding a maximum clique. In International conference on discrete mathematics and theoretical computer science (pp. 278-289). Springer, Berlin, Heidelberg.

Babel, L., & Tinhofer, G. (1990). A branch and bound algorithm for the maximum clique problem. Zeitschrift für Operations Research, 34(3), 207-217.

Plan

Rappel : problème de stable maximum pondéré (WMIS)

Pour tout sommet V d'un graphe $G = (V, E)$, on associe un poids $c_v \geqslant 0$. On cherche à sélectionner un sous ensemble de sommets indépendants $V' \subset V$ en maximisant le poids total de sommets sélectionnés dans V'.

Variables binaires $z_v \in \{0, 1\}$, où $z_v = 1$ si le sommet v est considéré dans le stable, c'est-à-dire choisi dans V'.

Cela donne la formulation PLNE suivante :

$$\begin{array}{lll} & & \max\limits_{z \in \{0,1\}^{|V|}} \sum\limits_{v \in V} c_v z_v \\ s.c : & \forall e = (v_1, v_2) \in E, & z_{v_1} + z_{v_2} \leqslant 1 \\ & \forall v \in V & z_v \in \{0, 1\} \end{array} \tag{7}$$

CF Partiel : se résout en temps polynomial si les degrés sont tous au plus deux, par programmation dynamique sur les composante connexes

Algorithme Branch&Reduce pour WMIS

WMIS se résout en temps polynomial si les degrés sont tous au plus deux, par programmation dynamique sur les composante connexes

idée : on se ramène à ce cas dans un parcours arborescent, dont les feuilles seront des graphes de degrés 2, et se résolvent par algo polynomial spécifique

Branchements : on décide qu'un noeud doit être sélectionné ou enlevé. Réduit le nombre d'arête du graphe restant.

On choisit pour brancher un noeud de degré maximal d. Si $d \leqslant 2$, on résout avec l'algorithme dédié. Sinon, on a deux branchements. :
- En imposant le sommet dans le stable, cela enlève ses voisins, $d \geqslant 3$ autres sommets
- En imposant le sommet hors du stable, cela abaisse le degré résiduel de ses $d \geqslant 3$ voisins

Un tel algorithme est dit Branch&Reduce.

Algorithme de complexité temporelle exponentielle dans pire cas, mais peut être implémenté avec une complexité polynomiale en mémoire (avec parcours DFS)

Branch&Reduce pour WMIS et complexité

Calculs “heuristiques” de complexité, pour donner les idées, se démontrent rigoureusement.

Soit $T(n)$ le temps pour résoudre tout problème WMIS à n sommets

Avec le branchement sur le sommet v de degré $d(v) \geqslant 3$:

$$T(n) \leqslant T(n-1) + T(n-1-d(v)) \leqslant T(n-1) + T(n-4)$$

Résoudre $T(n) = T(n-1) + T(n-4)$ avec $T(n) = x^n$ et $x > 0$ donne $x^n = x^{n-1} + x^{n-4}$ puis $x^4 = x^3 + 1$. La plus grande racine x est $x \approx 1,3803$

L'algorithme Branch&Reduce a une complexité en $\mathcal{O}^*(1,3803^N)$

Calcul et bornes peuvent être raffinées, cette version simple montre comment dériver une complexité exponentielle.

Rappels de résultats de complexité

Tout graphe de n sommets a au plus $3^{n/3} \approx 1.4422^n$ stables maximaux.

Moon, J. W. ; Moser, L. (1965), "On cliques in graphs", Israel Journal of Mathematics, 3 : 23–28.

C'est toujours mieux que $O(n^2 2^n)$ avec une énumération naïve "brute force".

Le calcul d'un stable maximum se fait en temps $\mathcal{O}^*(1.1996^n)$ avec un espace mémoire polynomial.

Xiao, M. et Nagamochi, H. (2017), "Exact algorithms for maximum independent set", Information and Computation, 255 : 126–146.

Sur des graphes de degré maximum 3, la complexité temporelle passe à $\mathcal{O}^*(1.0836^n)$.

Xiao, M. et Nagamochi, H. (2013), "Confining sets and avoiding bottleneck cases : A simple maximum independent set algorithm in degree-3 graphs", Theoretical Computer Science, 469 : 92–104.

Problème polynomial lorsque le degré maximal est 2 et sur des graphes spécifiques "claw-free graphs", "P5-free graphs", graphes parfaits et graphes cordaux (complexité linéaire dans ce cas).

Combiner Branch&Reduce et Branch&Bound ?

- Branch&Reduce : un schéma arborescent conçu pour minimiser la complexité de pire cas
- Branch&Bound : un schéma arborescent conçu pour minimiser les temps de calculs en moyenne, sans garantie de pire cas
- Ces différences majeures font qu'en général Branch&Reduce est plus lent que Branch&Bound, pas vraiment d'hybridation en général
- Cas du stable de pondération maximum, ça peut se rejoindre :
  - Branchements de Branch&Bound non discriminant, bcp d'égalités avec PL. Pourquoi pas avoir le critère Branch&Reduce de branchement ?
  - Terminaison avec graphe de degré 2 : la résolution PL aussi entière dans ce cas, par totale unimodularité
  - Bornes faciles à calculer et critère d'arrêt d'exploration de borne peuvent accélérer Branch&Reduce en pratique
  - Garde t'on le branchement DFS du Branch&Reduce pour avoir un espace mémoire de taille constante ?

Plan

Algorithme "Beam Search"

- Beam Search : algorithme de recherche en faisceau
- Idée : recherche arborescente est exponentielle (B&B, PPC), limiter le parcours en largeur à k noeuds, k paramètre.
- Choix des k noeuds : selon un critère glouton ou de borne.
- Partant de k noeuds, on considère les k meilleurs noeuds fils suivant le critère choisi précédemment.
- Variante : on peut garder une part d'aléatoire (comme GRASP) dans les solutions retenues
- N.B : peut être codé génériquement sur un algorithme de recherche arborescente Branch&Bound.
- Beam Search : méthode utilisée en IA pour limiter un parcours arborescent
- Beam Search : algorithme très efficace pour des problèmes fortement contraints (ex : Challenge ROADEF 2018)

Algorithme "Beam Search" : ex TSP

- Le noeud 1 est le premier choisi
- Initialisation : on choisit les k villes les plus proches de la ville 1, initialisation avec k listes de 2 villes.
- Pour chacun des k noeuds, on calcule toutes les distances en rajoutant une ville non incluse dans la sous-énumération du noeud. $O(Nk)$ sous noeuds énumérés, étape parallélisable.
- On considère les k meilleurs nouveaux noeuds fils, (ou $k' < k$ meilleurs et $k - k'$ choisis aléatoirement).
- On itère jusqu'à avoir des solutions réalisables.
- N.B : schéma permet d'élaguer des sous-énumération impossibles pour TSPTW, peut aussi être amélioré en détectant les conflits au plus tôt comme en PPC.

Références

• Libralesso, L. (2020). Recherches arborescentes anytime pour l'optimisation combinatoire (Doctoral dissertation, Université Grenoble Alpes).

• Libralesso, L., & Fontan, F. (2021). An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. European Journal of Operational Research, 291(3), 883-893.

• Libralesso, L., Focke, P. A., Secardin, A., & Jost, V. (2021). Iterative beam search algorithms for the permutation flowshop. European Journal of Operational Research. available as arXiv preprint arXiv :2009.05800.

• Parreño, F., Alonso, M. T., & Alvarez-Valdés, R. (2020). Solving a large cutting problem in the glass manufacturing industry. European Journal of Operational Research, 287(1), 378-388.

• Yavuz, M. (2017). An iterated beam search algorithm for the green vehicle routing problem. Networks, 69(3), 317-328.

• Tamannaei, M., & Irandoost, I. (2019). Carpooling problem : A new mathematical model, branch-and-bound, and heuristic beam search algorithm. Journal of Intelligent Transportation Systems, 23(3), 203-215.

Plan

IA et recherche arborescente

L'IA utilise des méthodes de recherche arborescente, avec des similarités à l'algorithme de Branch&Bound.

Point clé : limiter les énumérations

Solveurs SAT, Programmation par Contraintes (PPC) : des méthodes de l'IA traditionnelle. Le lien PLNE-PPC développé dans cette partie.

Elagage alpha-bêta : méthode arborescente pour des problèmes minimax. (grand classique de l'IA)

Algorithme A^* : accélération de l'algorithme de Dijkstra pour des plus court chemins, avec des idées similaires au Branch&Bound. (grand classique de l'IA aussi)

Beam Search (recherche par faisceaux) : heuristique pour restreindre une exploration arborescente, par nature exponentielle. (on en dira quelques mots au prochain cours, approche venue de l'IA, qui a récemment montré un grand intérêt en RO pour des problèmes d'optimisation combinatoire)

Problème de satisfaction de contraintes (CSP)

Un CSP se modélise formellement par un triplet (X, D, C) tel que :

$X = \{X_1, X_2, \ldots, X_n\}$ désigne l'ensemble des variables (à valeurs discrètes) du problème,

D est une fonction qui associe à chaque variable X_i son domaine $D(X_i)$, l'ensemble (fini) des valeurs que peut prendre X_i,

$C = \{C_1, C_2, \ldots, C_k\}$ est l'ensemble des contraintes.

Chaque contrainte C_j est une relation entre un sous ensemble de variables de X.

Question : existe-t'il une affectation des valeurs qui vérifie toutes les contraintes ?

Problème de décision

Résolution, programmation par contraintes

- La PPC est une recherche arborescente, suivant des branchements
- Le *filtrage* ou *propagation de contraintes* évite l'énumération exhaustive, en réduisant les domaines des variables.
- Branchements pour énumérer plusieurs cas possibles
- Les algorithmes de filtrages de la PPC sont en général moins gourmand que la résolution de PL.
- De même que pour le B&B, la paramétrisation efficace de la PPC tient d'un bon compromis à trouver entre le temps nécessaire au filtrage et son efficacité à réduire des solutions.

Modélisation, programmation par contraintes

- Des contraintes linéaires et le cadre PLNE sont inclus la PPC
- D'autres types de contraintes : AllDifferent(X,Y,Z) pour imposer des valeurs distinctes ...
- "Model-and-run", avec algorithmes de filtrage dépendant de l'écriture des contraintes
- Un premier modèle s'écrit facilement en PPC, optimiser l'efficacité demande de l'expertise

Problèmes d'applications classiques

- Illustration sur le Sudoku : algorithmes de propragation de contraintes
- Autre problème classique : les n reines dans un jeu d'échec.
- Recherche de motif dans un graphe : existe t'il un sous isomorphisme entre deux graphes donnés ?
- Existe t'il un tour cycle passant par tous les points dans un graphe non complet ?
- Existe t'il un stable/une clique de cardinal au moins k ?

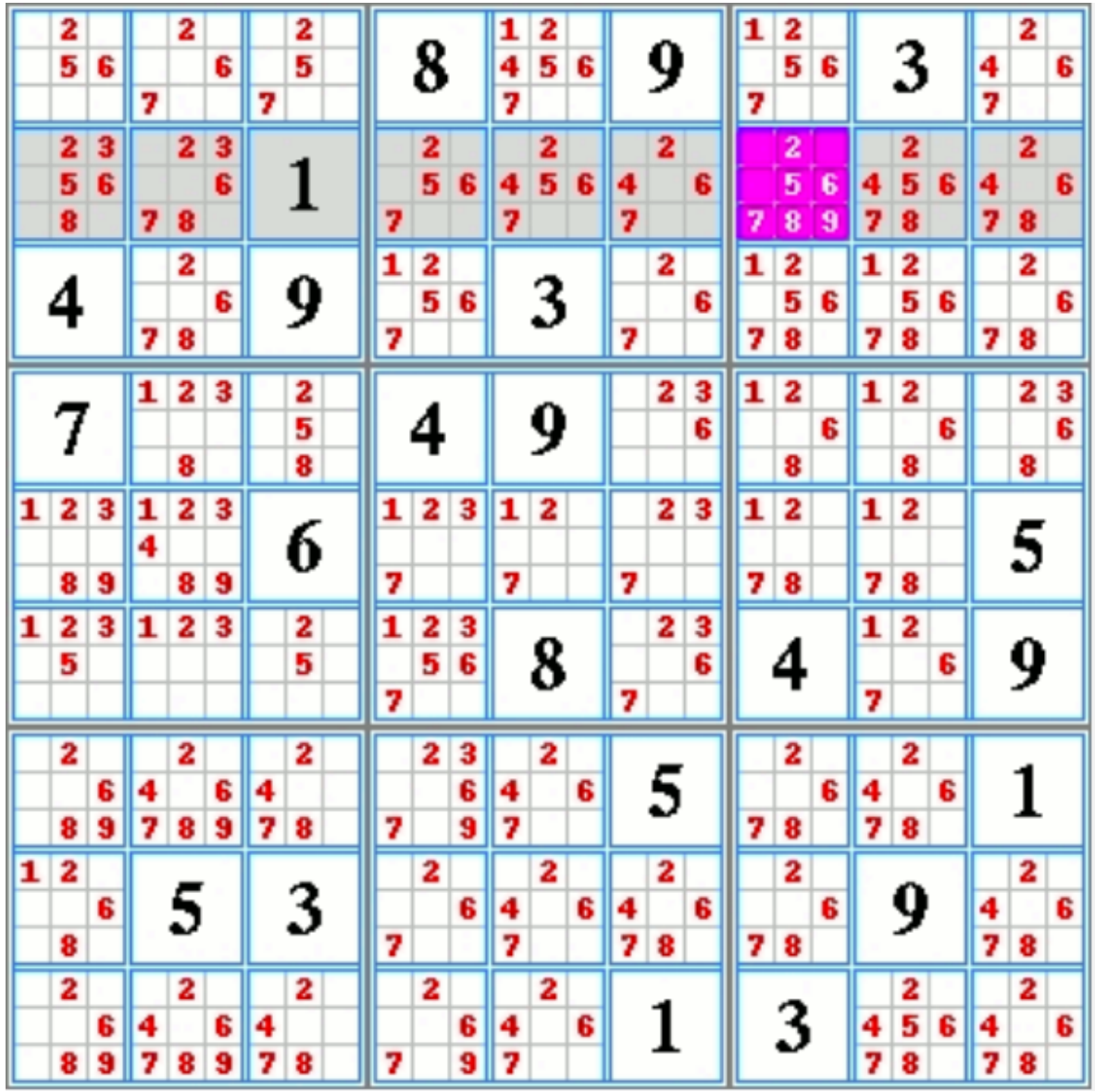
Modèle de Sudoku en PPC avec AllDifferent

- Variables et domaines :

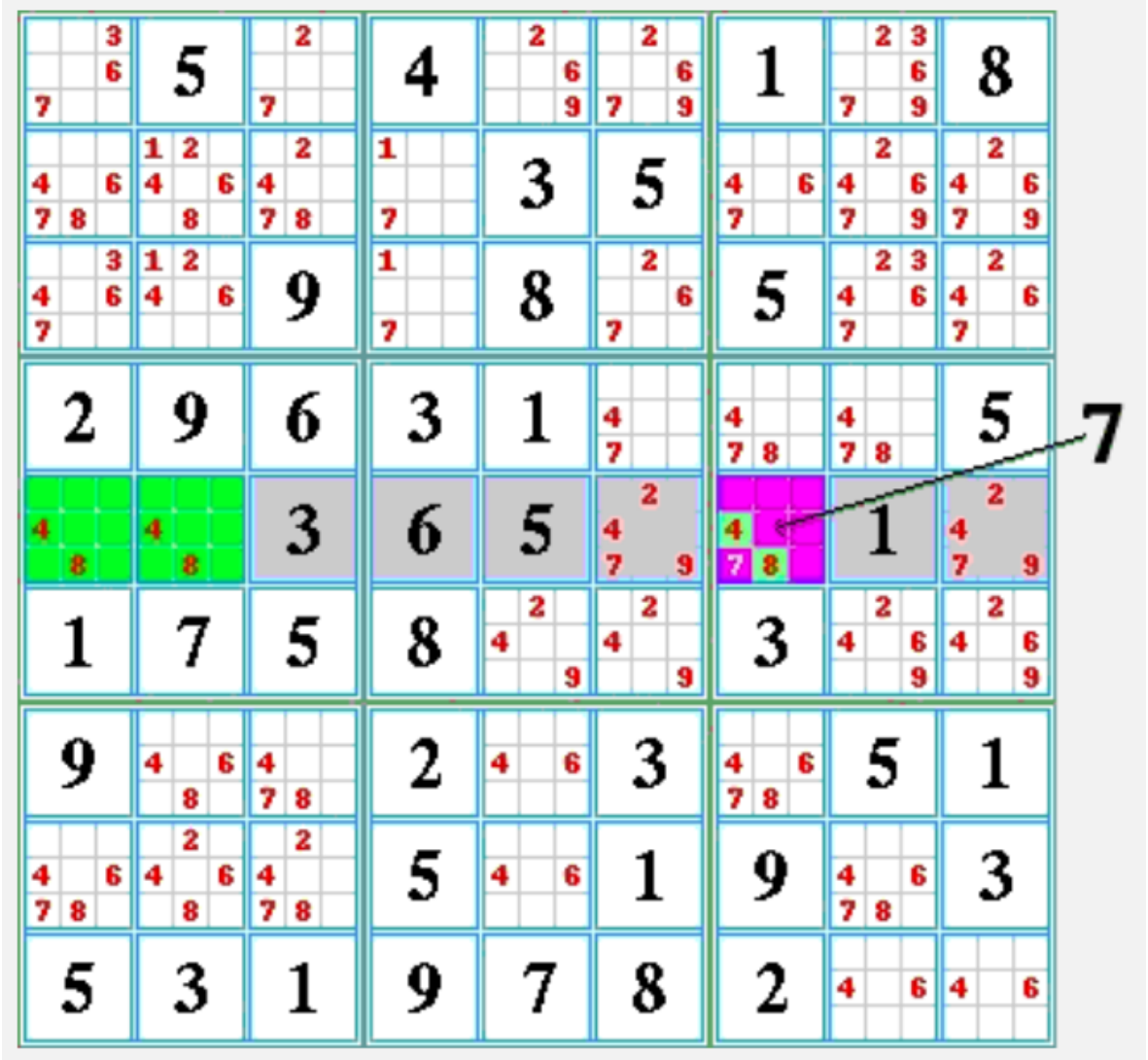
x[i,j] $\in$ 1..9 pour i et j allant de 1 à 9

- Contraintes :

Pour i allant de 1 à 9
    // valeurs différentes en ligne
    allDifferent(x[i,j] : j de 1 à 9)
    // valeurs différentes en colonne
    allDifferent(x[j,i] : j de 1 à 9);
Pour i allant de 0 à 2
  Pour j allant de 0 à 2
    // valeurs différentes dans les cases 3x3
    allDifferent(x[3*i+k,3*j+q] : k et q de 1 à 3);



Filtrage en propageant les valeurs définies pour réduire les domaines des autres variables



Filtrage plus avancé

Algorithme de retour sur trace (backtrack)

- Algorithme de retour sur trace (backtrack) : implémentation rapide de PPC pour le Sudoku.
- On avance dans la grille en allouant à chaque case la plus petite valeur possible vérifiant les contraintes d'unicité par lignes/colonnes/blocs avec les cases déjà allouées.
- Quand on aboutit à une erreur, on remonte à la précédente case en incrémentant la valeur de la case.
- On s'arrête quand on a trouvé une solution qui a pu remplir toute la grille.
- Si on a tout énuméré, pas de solution.
- Cadre PPC avec un filtrage moins fort que précédemment, avec un branchement lexicographique.
- Forte analogie avec schémas d'énumération et backtrack vu dans les cas TSP et MaxStable (juste le critère d'arrêt change)

Solveurs de PPC

- OR-tools (C++, Python) : à présent un excellent solveur de PLNE (le meilleur open source)

  `https://github.com/google/or-tools`

- Choco (Java).

- Gecode (C++)

- Cplex a son moteur de calcul de PPC : CPO. On peut utiliser OPL comme interface ou des API des langages usuels :

`https://github.com/IBMDecisionOptimization/docplex-examples/blob/master/examples/cp/basic/sudoku.py`

Code OPL du Sudoku en PPC avec AllDifferent

```
using CP;

int taille=9;
dvar int x[1..taille,1..taille] in 1..taille;

subject to {
    forall(i in 1..taille) {
        allDifferent(all(j in 1..taille) x[i,j]);
        allDifferent(all(j in 1..taille) x[j,i]);
    }
    forall(i in 0..2)
        forall(j in 0..2)
            allDifferent(all(k in 1..3, q in 1..3) x[3*i+k,3*j+q]);
x[1,1]==7;x[2,2]==2;x[2,3]==5;x[3,2]==6;x[3,3]==1;x[3,4]==9;x[1,6]==5;x[1,8]==3;x[2,9]==7;
x[5,1]==4;x[6,2]==7;x[5,4]==3;x[5,5]==2;x[6,6]==6;x[4,7]==9;x[6,8]==4;x[6,9]==2;
x[7,2]==4;x[7,3]==6;x[9,3]==7;x[8,5]==4;x[9,5]==5;x[8,7]==2;x[9,7]==1;
```

Autres exemples d'interfaces et de solveurs

OR-tools : `http://www.hakank.org/google_or_tools/`

Choco : `https://forgemia.inra.fr/degivry/mposc/-/blob/master/TP/TP2013/Sudoku.java`

module PPC de FICO Xpress : `https://examples.xpress.fico.com/example.pl?id=sudoku`

On peut aussi utiliser les solveurs de PLNE :

`https://towardsdatascience.com/using-integer-linear-programming-to-solve-sudoku-puzzles-15e9d2a70baa`

`http://profs.sci.univr.it/~rrizzi/classes/PLS2015/sudoku/doc/497_Olszowy_Wiktor_Sudoku.pdf`

Pour résumer la PPC, comme satisfaction de contraintes

- Modélisation globale : travail se concentre sur l'écriture des contraintes.
- "Model & run" avec un langage de modélisation plus souple que PLNE. mias pas de variables continues, que des quantités discrètes
- Résolution exponentielle, recherche arborescente avec branchements et backtrack.
- Questions de décisions, mais aussi recherche arborescente sur un critère d'arrêt pour explorer différentes solutions d'un problème d'optimisation.
- Cadre adapté pour résoudre des problèmes d'optimisation très contraints, où un algorithme de filtrage est efficace.
- Filtrage en PPC et relaxation/évaluation de critères d'arrêts en PLNE sont des mécanismes analogues pour stopper au plus vite une énumération partielle. Deux mécanismes qui peuvent être combinés :

Achterberg, T. (2004). SCIP-a framework to integrate constraint and mixed integer programming.
Achterberg, T. (2009). SCIP : solving constraint integer programs. Mathematical Programming Computation, 1(1), 1-41.

Utiliser un solveur PPC comme solveur de PLNE ?

- Si initialement la PPC était conçue pour des problèmes de décision, la recherche arborescente s'adapte avec une fonction objectif.
- On ne s'arrête plus quand on a trouvé une première solution réalisable. En temps limité, une heuristique parcourant des solutions.
- Le parcours de l'arbre de branchement peut être orienté par la fonction objectif.
- Avec CPO, on peut avoir la garantie d'optimalité.
- Un parcours de type PPC est en général moins gourmand en temps de calcul à chaque noeud, mais peut explorer plus de noeuds qu'en PLNE, un compromis à trouver.
- Filtrage en PPC très différent de la relaxation continue. Intéressant avec un filtrage PPC très adapté au problème.
- Mauvaise idée : utiliser la PPC sur le pb de sac à dos, la relaxation continue guide et limite très bien la recherche arborescente.

Exemples de PLNE où la PPC est efficace

- Avec un code concis, la PPC est assez efficace pour le pb de voyageur de commerce.
- TSP : une permutation des villes est un AllDifferent sur un vecteur de taille N avec N valeurs.
- Modélisation facile et concise, pour l'efficacité, des adaptations des algos de filtrages sont étudiés
- La PC est très efficace pour des problèmes de plannings (type emploi du temps) et ordonnancement
- Développement spécifiques dans CPO pour les structures temporelles.

Plan

Recherche arborescente

- Algorithme de Branch&Cut : pour des formulations avec ensemble de contraintes no énumérables. Outils de l'approche polyédrale
- Algorithme de Branch&Bound généralisé : pas besoin d'avoir une relaxation PL, n'importe quelle borne permet de définir un algorithme de type Branch&Bound.
- Algorithme de Branch&Reduce : un outil pour de la complexité exponentielle de pire cas.
- Liens avec la recherche arborescente de l'IA :
  - Mécanismes de bornes et d'élagage dans l'algorithme A^*, ou l'élagage alpha-bêta. (aussi avec certains cas de programmation dynamique).
  - Beam Search : limitation heuristique d'un parcours arborescent, frontière de l'IA et de la RO.
  - Résolution Programmation par Contraintes : famille d'algorithme de recherche arborescente a des similarités et des hybridations avec l'algorithme de Branch & Bound

Pour la suite

- TP de GLPK : savoir utiliser un modeleur (principes généraux) et faire le lien entre théorie/composants de l'algo de B&B avec les résultats pratiques obtenus.
- Une dernière recherche arborescente : Branch&Price, avec la relaxation Lagrangienne et l'algorithme de génération de colonnes