(Méta-)Heuristiques

1 Que faire face à un problème NP-dur ?

Supposons qu'on ait à résoudre un problème d'optimisation du type "trouver une solution qui optimise une fonction objectif" et qu'on ait prouvé que ce problème soit NP-dur : on doit donc abandonner l'idée d'avoir un algorithme polynomial qui donne à coup sûr la solution optimale, à moins de prétendre prouver que $P = NP$. On pourra alors avoir "grosso modo" deux attitudes : abandonner l'idée d'avoir un algorithme polynomial ou abandonner l'idée d'avoir un algorithme qui donne toujours la solution exacte.

- Dans le premier cas, on peut essayer d'obtenir un algorithme exact non polynomial mais optimisé et ne l'appliquer que sur des données relativement petites ou "faciles" : en effet, montrer qu'un problème est $NP-$dur indique seulement que "le pire des cas est dur".

  Pour certains problèmes, comme le problème du sac à dos, il existe des algorithmes pseudo-polynomiaux : un algorithme *pseudo-polynomial* est un algorithme qui est polynomial si les entiers de la donnée sont codés en base 1 ; par exemple, si la donnée est un entier n, un algorithme en $\Theta(n)$ n'est pas polynomial mais il est pseudo-polynomial. Ils peuvent être raisonnablement efficaces sous certaines conditions.

  De même, pour certains problèmes, des algorithmes donnent des bons résultats sur les données courantes qui ont une structure particulière. Le cadre de la complexité paramétrée permet d'analyser la complexité non pas seulement en fonction de la taille de la donnée mais en fonction de plusieurs paramètres. Dans ce cadre, on peut définir une notion d'algorithmes FPT (fixed-parameter tractable) pour certains problèmes, permettant d'avoir des algorithmes efficaces pour des problèmes NP-durs si le paramètre fixé a une valeur petite.

  On peut aussi raffiner les méthodes exhaustives en utilisant des méthodes de type "Branch and Bound" ou "séparation, évaluation" qui évitent de parcourir l'espace de toutes les solutions en "élaguant" (pruning), par exemple en évitant de développer une solution partielle si on est sûr qu'elle sera moins bonne qu'une solution déjà trouvée.

  On peut enfin utiliser des **solveurs** Sat [1] ou de programmation linéaire en entiers qui intègrent souvent beaucoup d'optimisations et sont efficaces sur des données de taille raisonnable.

- Dans le deuxième cas, on laissera donc tomber la contrainte "trouver la solution optimale". Dans ce cas, on utilise donc un *algorithme d'approximation* : il fournit une solution toujours correcte mais pas nécessairement optimale. Un algorithme d'approximation peut être déterministe - pour une entrée donnée, il donnera toujours la même solution (heuristiques [2] gloutonnes, optimum local, tabou....), ou stochastique (recuit simulé, algorithme génétique,...). Il peut être basé sur un critère glouton propre au problème - heuristique gloutonne - ou sur une méta-heuristique adaptée au problème.

2 Heuristiques gloutonnes

Le schéma d'une heuristique gloutonne est souvent similaire à celui d'un algorithme glouton exact, consistant à construire incrémentalement une solution :

```
//initialisation
Ens=E;
Solpartielle= ensemble (ou suite) "vide"
//calcul "glouton"
tant que Solpartielle non Solution ( et Ens non vide)
 select(x,Ens);        //on choisit x selon critère glouton
```

1. Daniel Le Berre, du laboratoire CRIL (Lens) vient d'obtenir la médaille d'innovation du CNRS pour le Solveur Sat4J
2. heuristique (du grec heuriskein, trouver) : technique ou art d'inventer, qui aide à faire des découvertes

```
 si Solpartielle + x  est une solution partielle alors  S=S + x; fsi;
 //dans certains problèmes, c'est toujours le cas
 Ens=Ens - x;            //x ne sera plus considéré
fin tant que;
```

La différence avec les algorithmes gloutons réside dans le fait qu'on n'impose plus que la solution obtenue soit optimale : on obtient donc un algorithme d'approximation.

Il existe de nombreuses heuristiques gloutonnes dédiées à des problèmes classiques : coloriage de graphes, affectation, mise en sachets,... certaines donnant de bons résultats et étant très utilisées.

3 Algorithmes d'approximation et Garanties

Un algorithme A d'approximation sera donc un algorithme qui donnera une solution toujours correcte, mais pas nécessairement optimale. On se place ici dans le cas déterministe. Pour estimer la qualité de l'approximation A, l'idée est de comparer la solution proposée par l'algorithme et la solution optimale.

Soit donc $Opt(I)$ la solution optimale pour l'instance I, $A(I)$ la solution produite pour I par l'algorithme A -on se place dans le cas déterministe, donc elle est bien définie- et f la fonction objectif à optimiser.

On définit différentes notions de garanties :

- Garantie absolue : c'est la borne sup de $|f(A(I) - f(Opt(I))|$ pour toutes les instances I du problème. Ce n'est bien sûr pas forcément fini ; cette notion est rarement utilisée.
- Ratio de garantie : ce sera le ratio maximum $\frac{f(A(I))}{f(Opt(I))}$ pour toute instance I si la fonction objectif est à minimiser, ou le ratio maximum $\frac{f(Opt(I))}{f(A(I))}$ pour toute instance I si la fonction objectif doit être maximisée. Le ratio est donc toujours supérieur ou égal à 1 -si il est égal à 1 c'est que l'algorithme est exact !- : plus ce ratio est proche de 1, meilleure est la garantie de l'approximation.

D'autres notions de garantie peuvent être définies, comme les garanties relatives, les garanties à la limite.

On retrouve le même problème que pour la complexité : *les garanties ne mesurent que le comportement dans le pire des cas et ne sont pas forcément révélatrices du comportement réel de l'approximation.* Autrement dit, un algorithme avec une garantie "mauvaise" peut donner un meilleur résultat "la plupart du temps" qu'un algorithme avec une très bonne garantie. Ceci dit, cela donne quand même une idée de la qualité de l'heuristique : si une heuristique a un ratio assez proche de 1, on peut estimer qu'elle est plutôt bonne !

Quel que soit le problème associé à un problème de décision NP-complet, peut-on toujours trouver un algorithme polynomial déterministe d'approximation avec une garantie ? Non ! Il y a plusieurs situations possibles :

- pour certains problèmes, on peut montrer que, quel que soit le ratio de garantie, on ne peut pas trouver d'algorithme polynomial qui offre ce ratio de "garantie", sauf si $P = NP$. C'est le cas du problème du voyageur de commerce, si on ne suppose pas que la distance vérifie l'inégalité triangulaire.
- pour certains problèmes, on connaît des algorithmes polynomiaux qui offrent une certaine garantie. C'est le cas du problème du voyageur de commerce, si on suppose que la distance vérifie l'inégalité triangulaire.
- pour d'autres, on montre même que quel que soit le ratio de garantie, on peut trouver un algorithme polynomial qui offre cette garantie. On a dans ce cas deux types de schémas : dans le cas le moins favorable, le degré du polynôme dépend de la qualité de l'approximation souhaitée. Dans l'autre cas, on a un schéma complet polynomial d'approximation i.e. on a un algorithme polynomial qui accepte en entrée une instance du problème et une qualité de solution souhaitée et ressort une solution ayant cette qualité. C'est le cas du Knapsack. En pratique, ce n'est pas forcément très intéressant car le degré du polynôme peut être grand.

4 Quelques mots sur quelques méta-heuristiques

Les méta-heuristiques sont des méthodes "génériques" pour rechercher une solution approchée aux problèmes d'optimisation. Une première famille est fondée sur la notion de parcours de l'ensemble de solutions, parcours basé sur une notion de voisinage. Une deuxième famille utilise une notion de population (de solutions), population qu'on fait évoluer en essayant de l'améliorer par rapport à l'objectif recherché.

4.1 Trois méthodes basées sur la notion de voisinage

L'idée sous-jacente est simple : on explore l'espace des solutions en partant d'un point et en se "déplaçant" de voisin en voisin : les voisins d'une solution seront des solutions proches de celle-ci, i.e. qu'on peut obtenir par exemple en transformant à peine la solution initiale. Par exemple, dans le problème du voyageur de commerce, on peut prendre comme voisins d'une tournée les tournées obtenues en supprimant deux arcs non contigus de la tournée et en reconnectant les deux sous-tournées obtenues. Pour appliquer les méthodes ci-dessous, il faut donc avoir au préalable défini une notion de voisinage d'une solution. Bien sûr, pour le même problème, plusieurs notions de voisinage peuvent être définies.

Recherche d'un optimum local (Hill-Climbing)

On part d'une solution si possible "bonne" (par exemple donnée par une heuristique ci-dessus) et on balaie l'ensemble des voisins de cette solution ; si il n'existe pas de voisin meilleur que notre solution, on a trouvé un optimum local et on arrête ; sinon, on choisit le meilleur des voisins et on recommence. Une autre implémentation consiste non pas à passer au meilleur des voisins à chaque étape mais au premier meilleur voisin trouvé. La convergence vers un optimum local pouvant être très lente, on peut éventuellement fixer un nombre de boucles maximum, si on veut limiter le temps d'exécution.

Cette méthode a l'inconvénient de pouvoir "rester bloquée" dans un optimum local : une fois un optimum local trouvé, on s'arrête, même si ce n'est pas l'optimum global. Selon le "paysage" des solutions, l'optimum local peut être très bon ou très mauvais par rapport à l'optimum global. Si la solution de départ est donnée par une heuristique déterministe, l'algorithme sera déterministe. Si elle est tirée au hasard, on a un algo non déterministe et donc plusieurs exécutions différentes sur la même instance pourront donner des solutions différentes et de qualités différentes.

La notion de voisinage est primordiale. Si les voisins sont très nombreux, on a de fortes chances de trouver l'optimum global mais visiter un voisinage peut être très long : on visitera une grande partie de l'espace des solutions. Si le voisinage est très restreint, on risque fort de rester bloqué dans un optimum local de "mauvaise qualité" : le choix de la notion de voisinage est un compromis entre efficacité et qualité. Les deux méthodes qui suivent peuvent être vues comme des méthodes de recherche locale avancées.

La méthode "Taboue"

A chaque étape, on recherche le meilleur voisin, mais on limite la recherche aux voisins non tabous : un voisin est a priori tabou si on a exploré cette solution durant les N précédentes itérations. La maintenance d'une liste taboue de voisins étant souvent coûteuse, on se borne souvent à stocker les transformations effectuées. A chaque itération, on choisit le meilleur voisin (ou un meilleur voisin selon le cas) correspondant à une transformation non taboue ; on effectue cette transformation, on la place dans la liste (file) taboue et on élimine la plus ancienne transformation de cette liste si la file est pleine. De plus, si la solution actuelle est la meilleure trouvée depuis le début, on la stocke. On s'arrête soit après un nombre fixé d'itérations, soit après un nombre fixé d'étapes n'ayant pas amélioré la solution. Pour une notion de voisin fixée, cette solution pourra être paramétrée par la longueur de la liste taboue.

La méthode "taboue" peut donc être vue comme une généralisation de la recherche d'optimum local (si N=0, c'est le hill-climbing) ; l'idée est de permettre de s'"échapper" d'un optimum local grâce à la liste taboue.

Le recuit simulé

La méthode (simulated annealing) est inspirée de la physique : le recuit est utilisé pour obtenir des états de faible énergie ; il est utilisé par exemple dans la fabrication du verre.

On part d'une solution quelconque (il n'est pas nécessairement intéressant de partir d'une "bonne" solution). A chaque étape, on tire au hasard une solution voisine. Si elle est meilleure, on l'adopte ; sinon on calcule l'augmentation de coût Δ ; on va adopter cette solution avec une probabilité liée à cette augmentation : plus le coût augmente, plus la probabilité de la garder sera faible. Mais cette probabilité dépend aussi du temps écoulé depuis le lancement de l'algorithme : au début, on accepte facilement un changement qui produit une solution plus coûteuse à la fin on l'accepte très difficilement. Plus précisément on calcule $accept = E^{-\Delta/T}$, T étant une quantité que nous appellerons *la température* (nous y reviendrons). On tire au hasard un réel p entre 0 et 1 : si p est inférieur à *accept*, on adopte la nouvelle solution, sinon on garde l'ancienne. Le schéma général est donc :

```
init(sol);
T:=T0;
loop
  sol'=voisin_au_hasard(sol);
  if meilleur (sol',sol) then sol=sol';
  else
    Delta=cout(sol') - cout(sol); --surcoût de sol'
    tirerproba(p); --p réel dans [0,1]
    if p<= e ^ (-Delta/kT)
    then sol=sol';
    --on accepte sol' avec proba accept
    --sinon on ne change pas
    end if;
  T:=refroidissement(T);
end loop;
```

Il faut donc choisir une température initiale, et pour assurer l'arrêt, une façon de diminuer la température au cours de l'exécution. La température initiale doit être assez élevée pour que *accept* soit assez grand au départ même si l'augmentation de coût est grande. Pour diminuer T, on peut par exemple multiplier T par un réel inférieur à 1 à chaque fois. Plus ce réel est proche de 1, plus le refroidissement est lent. On peut aussi raffiner : on effectue plusieurs cycles de recuit simulé, en repartant à chaque fois de la solution trouvée au cycle précédent. Une difficulté de cette heuristique réside dans le réglage du refroidissement ; celui-ci pourra être choisi différemment selon l'instance du problème.

4.2 Les méthodes à base de population

Une des méthodes les plus connues de cette famille est celle des algorithmes génétiques, créés par J. Holland (75) puis développés par David Goldberg et qui sont basés sur deux principes de génétique : la survie des individus les mieux adaptés et la recombinaison génétique. Contrairement aux algorithmes précédents qui essayaient d'améliorer un unique "individu-solution", les algorithmes génétiques font évoluer une *p*opulation de solutions. Le principe de base est de simuler l'évolution d'une population de solutions avec les règles citées ci-dessus en vue d'obtenir une solution ou un ensemble de solutions les plus adaptées. "A chaque génération, un nouvel ensemble de créatures artificielles est créé en utilisant des parties des meilleures solutions précédentes avec éventuellement des parties innovatrices" (Goldberg). Le schéma d'un algorithme génétique est donc :

```
créer la population initiale --par exemple au hasard
boucle
  sélectionner les individus à utiliser pour créer la prochaine population
  --sélection des plus adaptés
  hybridation des individus sélectionnés à  l'aide des opérateurs de croisement
  -- crossover
  -- production de chromosomes à partir de ceux des parents
  éventuellement effectuer des mutations aléatoires
  -- modification d'un gène (innovation)
fin boucle
```

L'arrêt se fera par exemple quand la population ne "s'améliore" plus de manière significative ou après un nombre fixé de générations. Pour la mise en œuvre il faut donc :

- choisir le codage (souvent binaire) d'une solution
- fixer la taille d'une population
- définir la fonction d'adaptation (fitness)
- définir les opérateurs de sélection, le principe étant : meilleure est notre adaptation, plus on a de chances d'être sélectionné
- définir les opérateurs de croisement
- définir éventuellement les opérateurs de mutation.

L'un des défauts de la méthode est encore la convergence prématurée vers une population "de mauvaise qualité".

Il existe d'autres méthodes basées sur des "populations", comme les algorithmes de colonies de fourmis ou d'abeilles ou l'optimisation par essaims particulaires.