RCP 106 : algorithmes approchés

Cédric BENTZ (CNAM)

MOCA B2 (RCP 106)
2016-2017

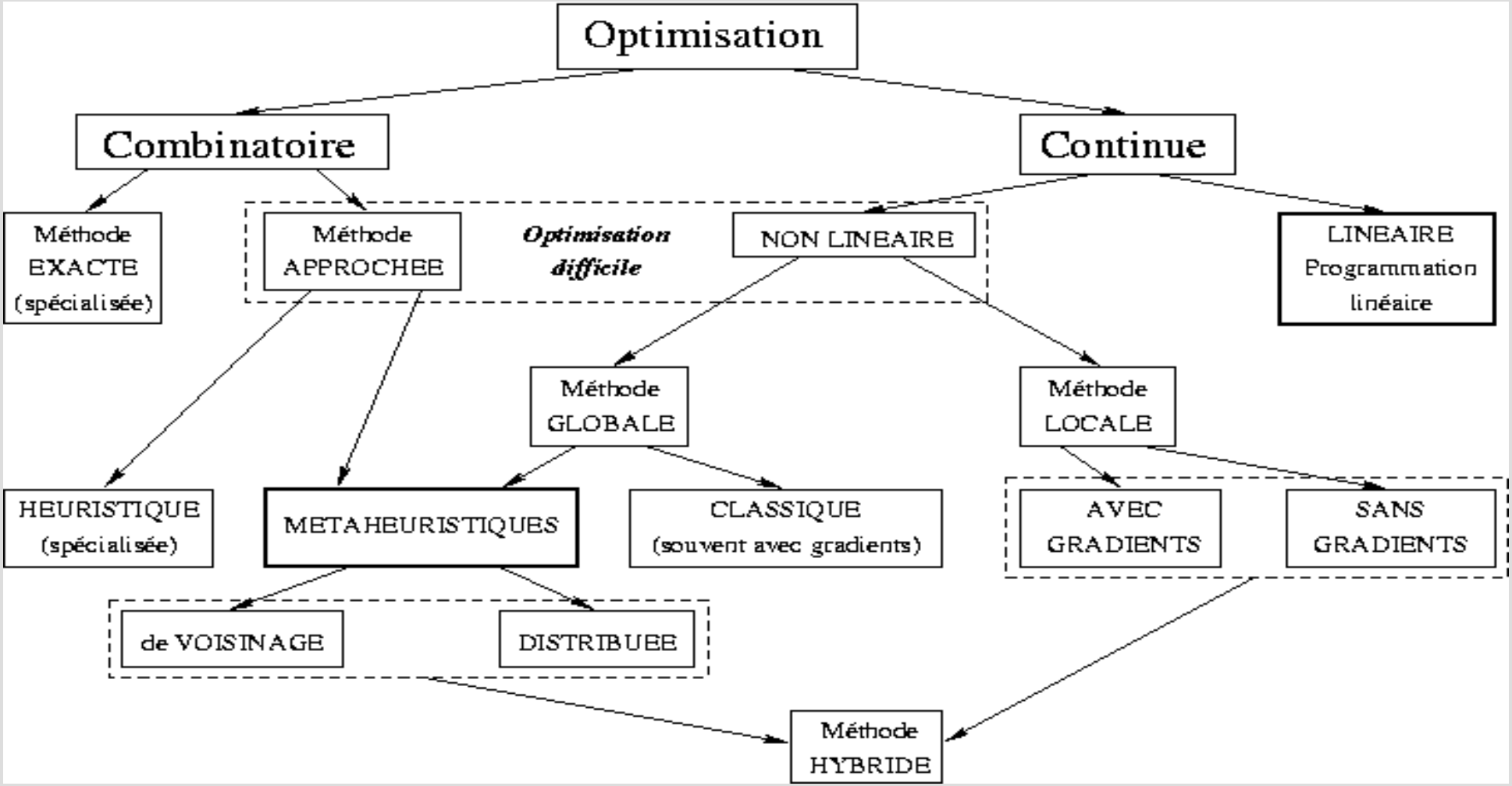
Contexte général

- On doit mener à bien un projet important, tout en minimisant le budget total alloué aux ressources
- Le problème se modélisant comme un problème d'optimisation combinatoire (OC) difficile, faut-il :
  - Attendre quelques jours (voire plus) pour espérer trouver une solution optimale (résolution exacte),
  - Attendre de quelques secondes à 5 minutes pour trouver une solution dont la valeur est 550 000 euros (la valeur optimale étant ici 500 000 euros),
  - Ou bien attendre quelques millisecondes pour trouver une solution qui « devrait » être bonne ?

Résolution « approchée » ?

- Trouver la solution optimale d'un problème d'OC peut prendre longtemps, très longtemps !
- Mais si on a besoin d'une solution rapidement ?
  - Idée : rechercher “rapidement” une “bonne” solution
    - Solution admissible,
    - Pas nécessairement optimale, mais meilleure que la plupart des autres solutions admissibles !
- De tel(le)s algorithmes et solutions sont dit(e)s approché(e)s, ou **heuristiques**

Panorama des méthodes de résolution



Algorithmes approchés

- Différentes méthodes à utiliser :
  - Méthodes par séparation & évaluation avortées
  - Algorithmes approchés spécifiques (à un problème)
    - Méthodes gloutonnes
    - Arrondi d'une solution continue optimale
    - Recherche locale
  - Méthodes génériques (métaheuristiques)
    - Recherche tabou, recuit simulé, algorithmes génétiques (ou évolutionnaires), colonies de fourmis, etc.

Méthodes par S&E avortées

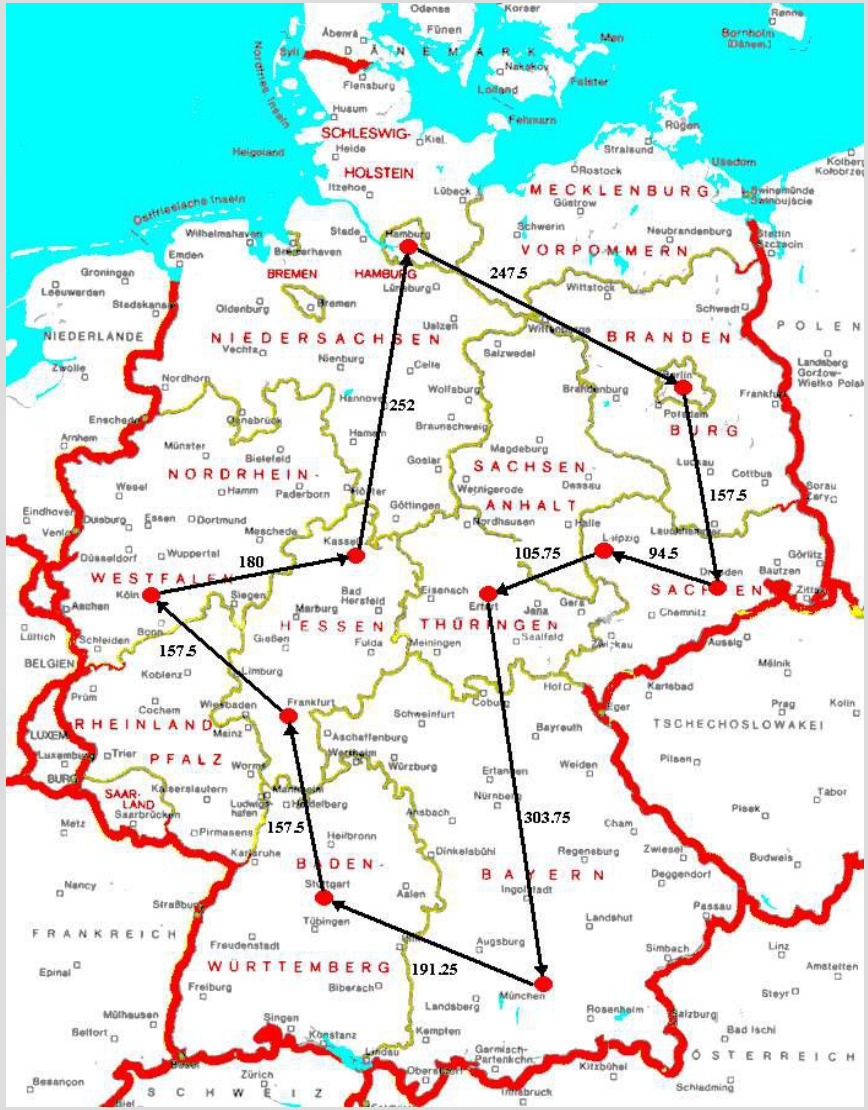
- Consistent simplement à exécuter partiellement une méthode par séparation & évaluation (S&E), et à l'arrêter quand un critère d'arrêt est vérifié :
  - Temps limite : risque = pas de solution admissible !
  - Ecart prédéfini entre bornes inférieure et supérieure
  - Critère mixte : temps limite si solution admissible OU écart prédéfini entre bornes inférieure et supérieure
- Avantage : réutilise les méthodes par S&E !
- Inconvénient : réutilise les méthodes par S&E !

Méthodes gloutonnes

- Principe général :
  - Définir un ordre sur les éléments à choisir (par exemple, en les triant selon un certain ordre),
  - Puis, les parcourir dans cet ordre : sélectionner un élément si un critère donné est vérifié (par exemple, si la solution en construction reste admissible).
- Avantages :
  - Algorithme simple et rapide,
  - Parfois optimal (exemple : Kruskal).

Exemple de méthode gloutonne : le voyageur de commerce (1/3)

- **Contexte** : un représentant de commerce (allemand) doit effectuer sa tournée de n villes, consistant à visiter chacune des villes une et une seule fois
- **Objectif** : minimiser la distance totale parcourue
- **Modèle** (graphe) : graphe dont les arêtes sont pondérées par des distances, et dont on cherche à parcourir tous les sommets une et une seule fois (+ retour au point de départ) en minimisant le poids total des arêtes de parcours utilisées

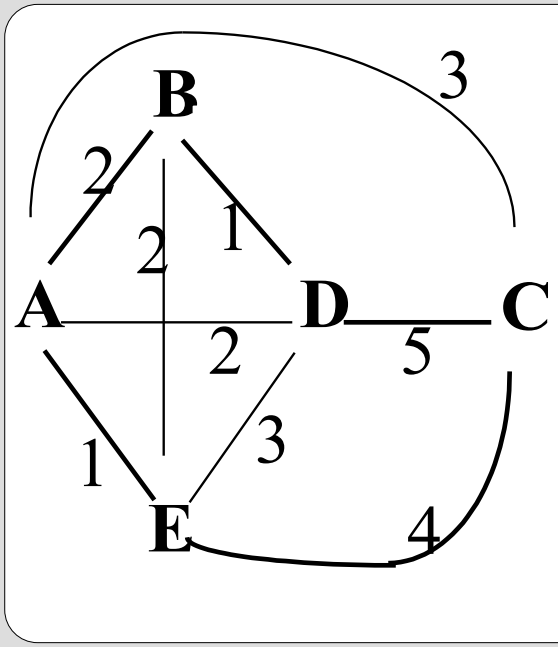


Exemple de méthode gloutonne : le voyageur de commerce (2/3)

- Problème NP-difficile
- Divers algorithmes gloutons (approchés) connus
- Exemple d'algorithme glouton (approché) :
  - Trier les arêtes par ordre de poids croissants,
  - Parcourir ensuite les arêtes dans cet ordre :
    - Sélectionner une arête si son ajout ne crée ni sous-cycle (cycle ne contenant pas tous les sommets du graphe) ni sommet de degré 3.

Exemple de méthode gloutonne : le voyageur de commerce (3/3)

- Exemple d'application de cet algorithme glouton :



([A,E],[B,D],[A,B],
~~[A,D]~~,~~[B,E]~~,~~[A,C]~~,
~~[E,D]~~,[C,E],[C,D])

[A,B,D,C,E,A]
valeur 13

(opt=12 [ACEBDA])

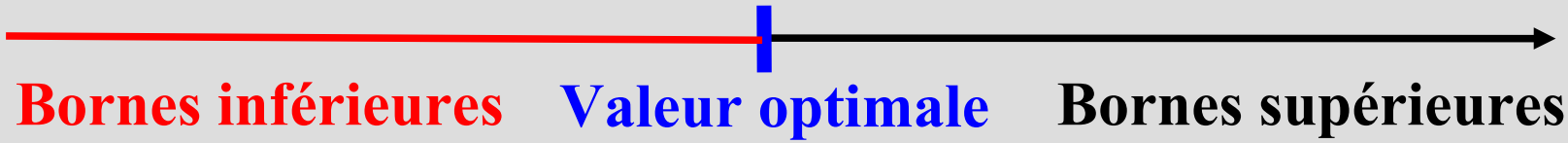
- Solution non optimale (valeur optimale = 12) !

Garantie de performance d'un algorithme approché (1/2)

- Peut-on prévoir, avant d'utiliser un algorithme approché, à quel point la solution calculée sera “proche” de la solution optimale ?
- Parfois, oui : notion de **garantie de performance**
- Ecart (ratio) entre les 2 valeurs **dans le pire cas** :
  - Maximisation : pire ratio possible entre valeur solution optimale et valeur solution approchée
  - Minimisation : pire ratio possible entre valeur solution approchée et valeur solution optimale

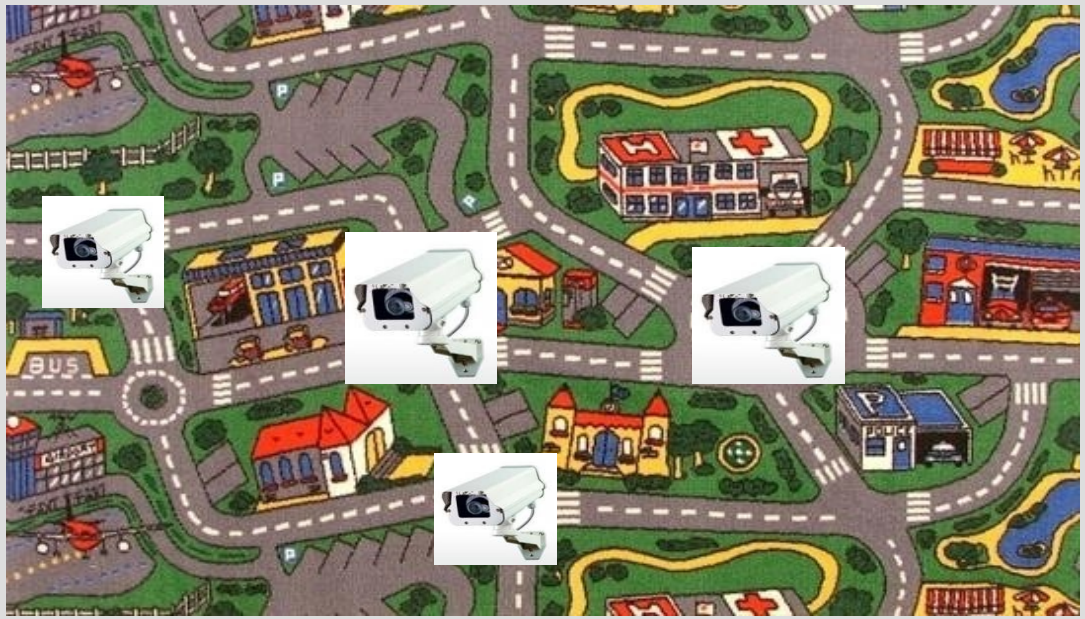
Garantie de performance d'un algorithme approché (2/2)

- Principales difficultés pour évaluer un tel ratio :
  - Comment identifier le pire cas ?
  - Valeur d'une solution optimale non connue !
    - En général, on utilise donc une borne (inf. si on minimise, sup. si on maximise) à la place.
    - Nécessité d'identifier une telle borne, puis d'en calculer la valeur (ou au moins de l'estimer).



Algo. approché glouton pour la couverture par les sommets (1/4)

- **Contexte** : placer le moins possible de caméras de surveillance à des carrefours pour surveiller toutes les rues d'un quartier
- **Formalisation** : on représente le quartier par un graphe, dont les sommets sont les carrefours et les arêtes les rues entre ces carrefours (on cherche alors un ensemble de sommets de taille minimum qui « couvre » toutes les arêtes du graphe)
- **Problème NP-difficile**



Algo. approché glouton pour la couverture par les sommets (2/4)

- Tant qu'il reste des arêtes non couvertes faire :
  - Choisir une telle arête (ordre arbitraire), et sélectionner dans la solution ses 2 extrémités.
  - A chaque fois qu'un sommet est sélectionné, supprimer les arêtes qui y sont incidentes.
- Temps d'exécution polynomial ?
  - Nombre d'itérations borné par le nombre d'arêtes !
- La solution construite est bien admissible, mais **seulement à la fin de l'algorithme**.

Algo. approché glouton pour la couverture par les sommets (3/4)

- Ratio d'approximation ?
  - Chaque arête considérée par l'algorithme doit de toute façon être couverte au moins une fois.
  - Borne inférieure sur la valeur optimale = taille d'un ensemble d'arêtes sans sommets en commun.
  - La valeur de la solution calculée est égale à **deux fois** cette taille ==> algorithme 2-approché !
  - En d'autres termes, la valeur de la solution calculée est au plus 2 fois celle d'une solution optimale (garantie de performance au pire cas).

Algo. approché glouton pour la couverture par les sommets (4/4)

- La valeur de la solution calculée peut en effet être deux fois celle d'une solution optimale :

- La solution calculée peut, en fonction des choix (arbitraires) effectués par l'algorithme, être optimale (si on a de la chance !) ou non :

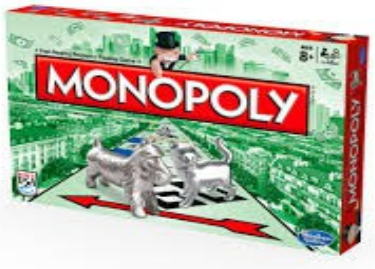
Algorithme approché glouton pour le sac-à-dos (1/6)

- **Contexte** (exemple) : on doit décider, parmi plusieurs marchandises, lesquelles stocker dans un container, de façon à maximiser le revenu qui y est associé
- **Formalisation** : chaque objet (marchandise) est muni d'un encombrement v_i et d'un revenu r_i, le volume total du sac (container), à ne pas dépasser, est V, et on cherche à maximiser le revenu total
- **Problème NP-difficile (dit problème du sac-à-dos)**

V m^3

?

?



r_1 \$

v_1 m^3

r_2 \$

v_2 m^3

Algorithme approché glouton pour le sac-à-dos (2/6)

- Solution optimale pour le problème du sac-à-dos si les objets peuvent être “fractionnés” ?
  - En triant (puis en sélectionnant) les objets par revenus décroissants ?
  - NON : un objet est plus intéressant qu'un autre s'il rapporte beaucoup d'argent (en $) par m³ !
  - En effet, si $r_i/v_i > r_j/v_j$, alors, pour tout q>0, placer un volume fixe de q m³ d'un objet dans le sac :
    - Rapporte $q*r_i/v_i$ (en $) pour i,
    - Rapporte $q*r_j/v_j<q*r_i/v_i$ (en $) pour j.

Algorithme approché glouton pour le sac-à-dos (3/6)

- Solution optimale pour le problème du sac-à-dos si les objets peuvent être “fractionnés” (suite) ?
  - Trier les objets par ordre des r_i/v_i décroissants,
  - Les parcourir dans cet ordre, et placer dans le sac la plus grande quantité possible de chaque objet.
- Ainsi, tout objet sera :
  - Soit intégralement dans le sac,
  - Soit intégralement hors du sac,
  - Soit en partie dans le sac (pour au plus un objet).

Algorithme approché glouton pour le sac-à-dos (4/6)

- Un tel ordre est-il également un “bon” ordre pour un algorithme glouton lorsque les objets ne peuvent être “fractionnés” ?
  - Parcourir les objets dans cet ordre, et placer un objet dans le sac si le volume restant est suffisant.
  - On considère 2 objets : l'un de revenu 1 et volume 1, l'autre de revenu V-1 et de volume V.
  - On remarque que 1/1 > (V-1)/V.
  - La solution renvoyée par l'algorithme glouton est de valeur 1, alors que la valeur optimale est V-1 !

Algorithme approché glouton pour le sac-à-dos (5/6)

- Meilleure idée : garder la meilleure des solutions obtenues par deux algorithmes gloutons !
  - Première solution : la solution précédente
  - Deuxième solution : trier les objets par revenus décroissants, et les sélectionner dans cet ordre
- Solution admissible (durant tout le calcul), facile à calculer rapidement (en temps polynomial)

Algorithme approché glouton pour le sac-à-dos (6/6)

- Ratio d'approximation ?
  - Borne supérieure sur la valeur optimale = valeur d'une solution optimale lorsque les objets peuvent être “fractionnés”.
  - Valeur de cette borne = valeur première solution + fraction (éventuellement nulle) du revenu d'un des objets (= au plus valeur deuxième solution)
  - Ainsi, valeur solution calculée = au moins **la moitié** de la valeur optimale ==> algorithme 2-approché (garantie de performance au pire cas)

Méthodes par arrondi d'une solution continue optimale (1/2)

- Principe général :
  - Modéliser le problème à résoudre comme un programme linéaire en nombres entiers (PLNE)
  - Le résoudre comme si les variables n'étaient pas entières (on obtient un programme linéaire)
  - Construire une solution entière à partir de la solution continue optimale obtenue, en dégradant le moins possible la valeur de la fonction objectif
  - La borne (inf. ou sup.) sur la valeur optimale est alors la valeur d'une solution continue optimale

Méthodes par arrondi d'une solution continue optimale (2/2)

- Avantages :
  - Réutilise les méthodes de résolution connues en programmation linéaire (PL),
  - Fournit d'office une borne (inf. ou sup.) sur la valeur optimale, via la programmation linéaire,
  - Si, par chance, la solution obtenue après résolution du programme linéaire est entière, alors c'est en fait une solution optimale pour le problème considéré !

Algo. approché par arrondi pour la couverture par les sommets (1/2)

- Formulation PLNE :
  - Une variable 0-1 x_i par sommet i : $x_i = 1$ si le sommet i est sélectionné dans la solution, 0 sinon
  - Fonction objectif : minimiser le nombre de sommets sélectionner = minimiser la somme des variables
  - Contraintes : pour toute arête i-j, $x_i + x_j \geq 1$
  - Si variables non entières (PL) : $x_i \in [0,1]$ pour tout i

Algo. approché par arrondi pour la couverture par les sommets (2/2)

- On résout le PL obtenu par un solveur (simplexe)
- Solution entière construite :
  - On pose $x_i=1$ si $x_i \geq 0.5$ dans la solution continue optimale, et $x_i=0$ sinon.
  - La solution construite est admissible, car, pour toute arête ij, il est impossible d'avoir $x_i<0.5$ et $x_j<0.5$
  - Valeur de la solution entière obtenue = au plus 2 fois celle de la solution continue optimale (pire cas = passer de 0.5 à 1) ==> algorithme 2-approché

Algo. approché par arrondi pour le problème du sac-à-dos (1/3)

- Formulation PLNE :
  - Une variable 0-1 x_i par objet i : x_i = 1 si l'objet i est inclus dans la solution (mis dans le sac), 0 sinon
  - Fonction objectif : maximiser la somme des valeurs des objets sélectionnés (mis dans le sac)
  - Contrainte : la somme des volumes des objets mis dans le sac ne doit pas dépasser V
  - Si variables non entières (PL) : $x_i \in [0,1]$ pour tout i

Algo. approché par arrondi pour le problème du sac-à-dos (2/3)

- On résout le PL obtenu par un solveur (simplexe)
  - Au plus une variable x_i non entière
- Solution entière construite :
  - On pose $x_i=0$ si $x_i<1$ dans la solution continue optimale, et $x_i=1$ sinon.
  - Les valeurs des x_i n'augmentent pas, donc cette solution entière est admissible par construction.
  - Cette solution entière “remplace” la première des 2 solutions utilisées dans l'approche gloutonne.

Algo. approché par arrondi pour le problème du sac-à-dos (3/3)

- Cette solution entière n'est pas nécessairement la même que la première des deux solutions utilisées dans l'approche gloutonne
- Ainsi, si r=(15,18,4,2,5,1), v=(3,4,1,1,3,1) et V=5 :
  - Glouton : $x_1=x_3=x_4=1$ et $x_2=x_5=x_6=0$, de valeur 21
  - Arrondi : $x_1=1$ et $x_2=x_3=x_4=x_5=x_6=0$, de valeur 15
- Pourtant, la même analyse permet de montrer que l'algorithme obtenu est un algo. 2-approché !

Algorithmes approchés par améliorations successives

- Glouton = la solution est construite au fur et à mesure, sans jamais remettre en cause les choix effectués à chaque étape
- Par arrondi d'une solution continue = on construit la solution une fois pour toutes à partir d'une autre
- Dans les 2 cas, une seule solution est considérée
- Peut-on, à l'inverse, obtenir une “bonne” solution approchée par un processus itératif (en parcourant ainsi un ensemble de solutions admissibles) ?

Voisinage d'une solution (1/2)

- ==> Autre idée pour trouver une “bonne” solution : essayer d'améliorer une solution initiale en la modifiant de façon itérative, grâce à des transformations élémentaires pré-définies
- A chaque itération, on transforme donc une solution en une autre solution qui lui est “proche”
- On définit ainsi une notion de **voisinage d'une solution admissible** donnée x = ensemble des solutions *admissibles* obtenues à partir de x à l'aide d'une seule transformation élémentaire

Voisinage d'une solution (2/2)

- Application à la couverture par les sommets :
  - Exemples de transformations élémentaires ?
    - Ajouter un sommet dans la couverture,
    - Retirer un sommet de la couverture,
    - Echanger deux sommets,
    - Etc.
- Application au problème du sac-à-dos :
  - Exemples de transformations élémentaires ?
    - Ajouter un objet dans le sac ou en retirer un,
    - Echanger deux objets,
    - Etc.

Recherche locale

- Grâce à la notion de voisinage, on définit celle de “recherche locale” à partir d'une solution x :
  - Parcourir le voisinage de x à la recherche d'une meilleure solution admissible, puis recommencer à partir de cette nouvelle solution, etc.
  - Plus le voisinage est étendu, plus cette recherche prendra du temps !
- Qu'obtient-on à la fin d'une recherche locale ?

Optimalité locale

- Une solution admissible est **localement optimale** si aucun de ses voisins n'est meilleur qu'elle
  - A partir de toute solution initiale, on peut aboutir par recherche locale à une solution localement optimale, au bout d'un certain temps
  - La solution initiale peut être n'importe quelle solution admissible (aléatoire, par S&E avortée, par algorithme glouton, par arrondi, etc.)
  - Solution localement optimale = **optimum local**
  - Par opposition, une solution optimale pour un problème donné est appelée **optimum global**

Optimalité locale vs globale (1/3)

- Par définition, un optimum global est un optimum local (quel que soit le voisinage)
- Par contre, un optimum local n'est pas nécessairement un optimum global !
  - Exemple de sac-à-dos ($x_i \in \{0,1\}$ pour tout i) :

$\max 15x_1 + 18x_2 + 4x_3 + 2x_4 + 5x_5 + x_6$

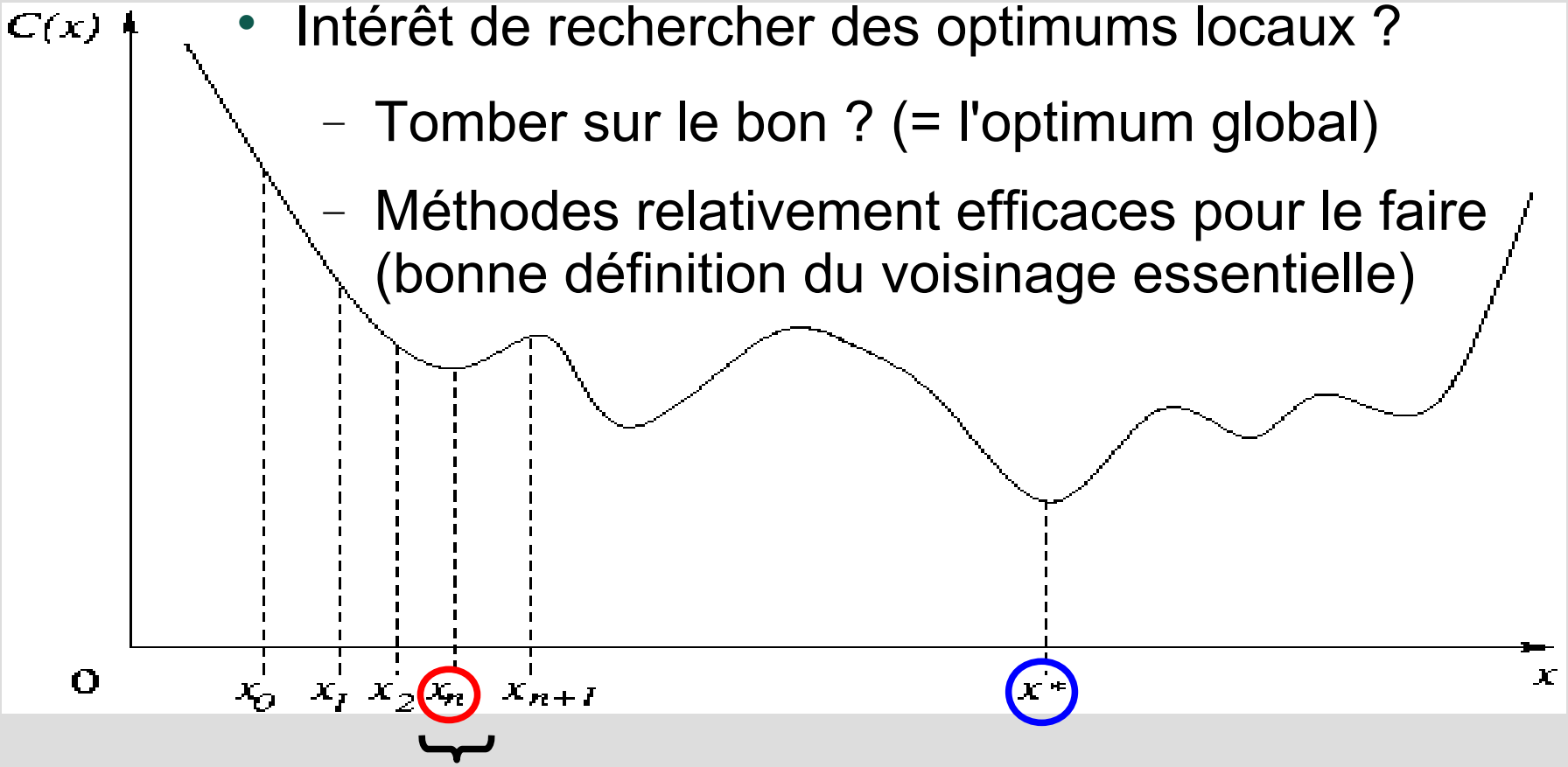
$3x_1 + 4x_2 + x_3 + x_4 + 3x_5 + x_6 \leq 5$

  - Transformations élémentaires (voisinage) : cf page 31

($x_1=x_3=x_4=1$, $x_2=x_5=x_6=0$) = optimum local, pas global (22 par S&E) !

Optimalité locale vs globale (2/3)

- Intérêt de rechercher des optimums locaux ?
  - Tomber sur le bon ? (= l'optimum global)
  - Méthodes relativement efficaces pour le faire (bonne définition du voisinage essentielle)



"Piège" potentiel : atteindre un optimum **local**, loin du **global** (et bien moins bon)

Optimalité locale vs globale (3/3)

- Dans certaines situations, un optimum local est **toujours** un optimum global :
  - Cas “convexe”,
  - Cas “linéaire” (problème convexe particulier), autrement dit **Programmation Linéaire**
- Selon la solution initiale considérée, l'optimum local obtenu à la fin peut être différent :
  - Stratégie pour améliorer la solution obtenue = faire de la recherche locale à partir d'un ensemble de plusieurs solutions admissibles initiales

Aller plus loin que la recherche locale ?

- Problématique :
  - Trouver un moyen de s'extraire d'un optimum local qui n'est pas un optimum global ?
  - Différentes stratégies pour éviter de tomber dans un tel piège ou de rester bloquer (perturber la solution, autoriser ponctuellement une dégradation de la valeur de la solution, etc.)
- Mise en oeuvre de ces stratégies “autour” d'une recherche locale = métaheuristique

Métaheuristiques

- Heuristique = algorithme approché
- Métaheuristique = algo. approché « générique »
  - Essentiellement : recherche locale + stratégie(s) pour s'extraire du voisinage d'un optimum local
- Plusieurs types : recuit simulé, recherche tabou, algorithmes génétiques, colonies de fourmis...
- Schéma commun :
  - Un algorithme générique (souvent) stochastique
  - Un ensemble de **paramètres**, qui permet d'adapter l'algorithme à chaque problème particulier

S'extraire d'un optimum local ?

- S'autoriser à dégrader ponctuellement la valeur de la solution (méthodes par recherche locale)
  - Avec une certaine probabilité, à contrôler
    - Recuit simulé
  - Si, après « apprentissage » (effet mémoire), aucune autre transformation n'est possible
    - Recherche tabou
- Faire évoluer plusieurs solutions en parallèle (et donc explorer simultanément plusieurs zones)
  - Algorithmes distribués/évolutifs (génétiques, etc.)

Quelques métaheuristiques en 2 mots et 7 slides (1/7)

- Recuit simulé (origines et présentation) :
  - Analogie avec la méthode du recuit en métallurgie : on chauffe un métal à haute température, puis on le refroidit lentement par paliers, pour obtenir un état d'énergie stable (= optimum local, qu'on espère global, de la fonction d'énergie)
  - Le recuit simulé utilise donc un paramètre appelé « température », qui diminue au cours du temps
  - Méthode proposée en 1983 par Kirkpatrick et al.

Quelques métaheuristiques en 2 mots et 7 slides (2/7)

- Recuit simulé (principes) :
  - S'extraire des optimums locaux ?
    - Durant la recherche locale, on s'autorise à dégrader la valeur de la solution, avec une certaine probabilité
    - Cette probabilité dépend de la température : plus la température est haute (au début), plus elle est haute
  - Réglage des paramètres ?
    - Température initiale (règle de Kirkpatrick)
    - Décroissance de la température entre deux paliers
    - Nombre de paliers et durée d'un palier
    - Solution initiale et conditions d'arrêt

Quelques métaheuristiques en 2 mots et 7 slides (3/7)

- Recherche taboue (origines et présentation) :
  - Aussi appelée méthode taboue
  - Méthode proposée par Glover en 1986
  - Ajoute un mécanisme de mémoire (de taille limitée) à la recherche locale
  - Ainsi, on tient à jour une liste (dite liste **taboue**) des transformations effectuées durant la recherche locale, et les transformations qui se trouvent dans cette liste sont « interdites »
  - La liste étant de taille limitée, ces transformations ne sont pas interdites pour toujours

Quelques métaheuristiques en 2 mots et 7 slides (4/7)

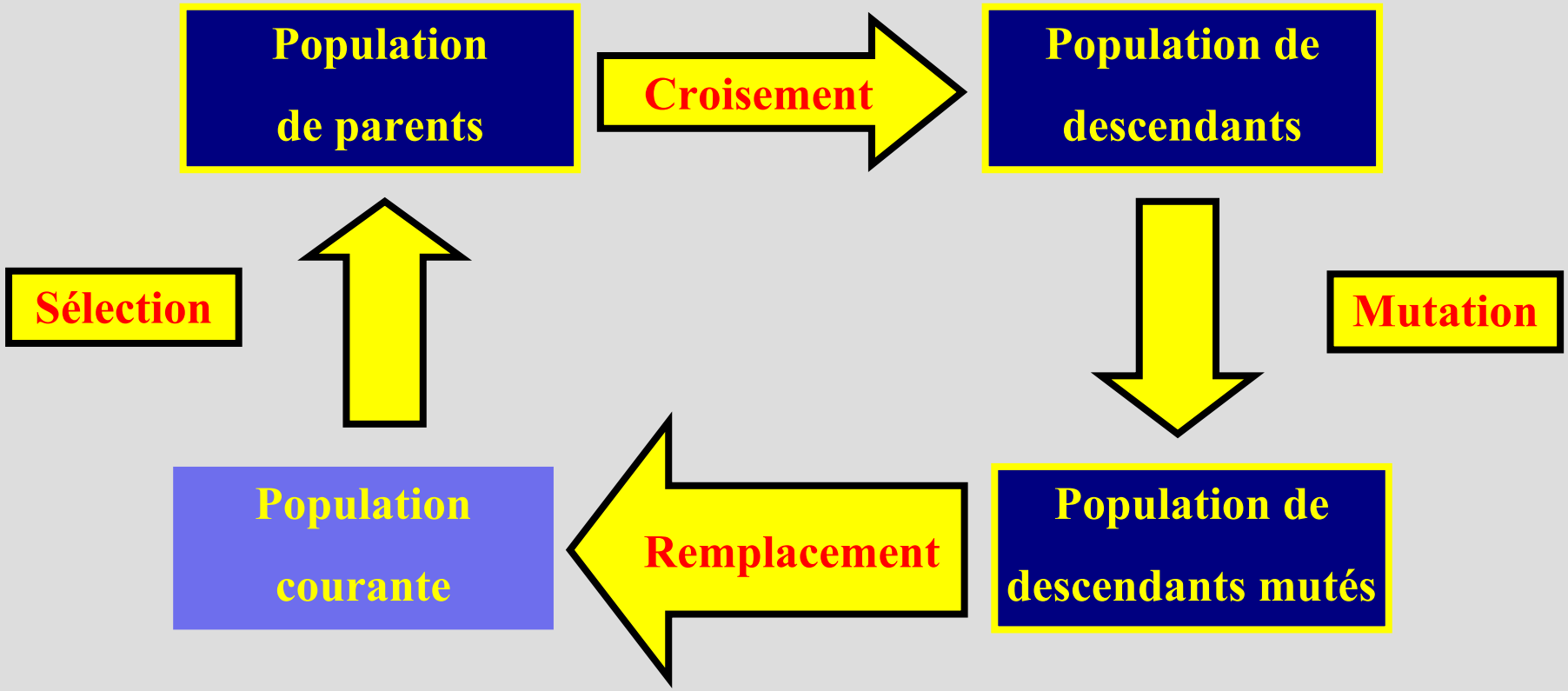
- Recherche taboue (principes) :
  - S'extraire des optimums locaux ?
    - Durant la recherche locale, on identifie le meilleur voisin obtenu par une transformation non interdite
    - Si ce meilleur voisin a une moins bonne valeur que la solution, on se déplace quand même vers lui, et cela peut permettre d'échapper à un optimum local
  - Réglages des paramètres ?
    - Taille de la liste taboue
    - Solution initiale (déterministe ?) et conditions d'arrêt
    - Méthode « moins stochastique » que le recuit simulé

Quelques métaheuristiques en 2 mots et 7 slides (5/7)

- Algorithmes génétiques/évolutionnaires (en bref) :
  - Analogie avec la théorie de l'évolution : croiser des « bons » gènes engendre de « bons » enfants
  - Pas basés sur une « vraie » recherche locale
  - Chaque individu (solution) est représenté par un code génétique (binaire)
  - A chaque itération, on a une population d'individus (solutions) de taille déterminée, et on croise ces individus pour engendrer une nouvelle génération

Quelques métaheuristiques en 2 mots et 7 slides (6/7)

- Algorithmes génétiques/évolutionnaires (suite) :



Quelques métaheuristiques en 2 mots et 7 slides (7/7)

- Algorithmes génétiques/évolutionnaires (fin) :
  - S'extraire d'un optimum local ?
    - On fait évoluer toute une population d'individus (solutions) en parallèle
  - Réglages des paramètres ?
    - Codage des solutions
    - Tailles des populations
    - Taux de croisement et de mutation
    - Population initiale et conditions d'arrêt

Quelle métaheuristique choisir ?

- Quelle est la plus efficace ?
  - Aucune, et toutes à la fois : ça dépend du problème !
  - En « moyenne », toutes ont la même efficacité : seul résultat théorique réellement applicable
- Laquelle choisir, pour un problème donné ?
  - Pas de règle absolue pour :
    - Choisir une métaheuristique,
    - Régler ses paramètres.
  - Expérience (savoir-faire) et/ou expérimentations (comparaison expérimentale) !

Mise en oeuvre de métaheuristiques

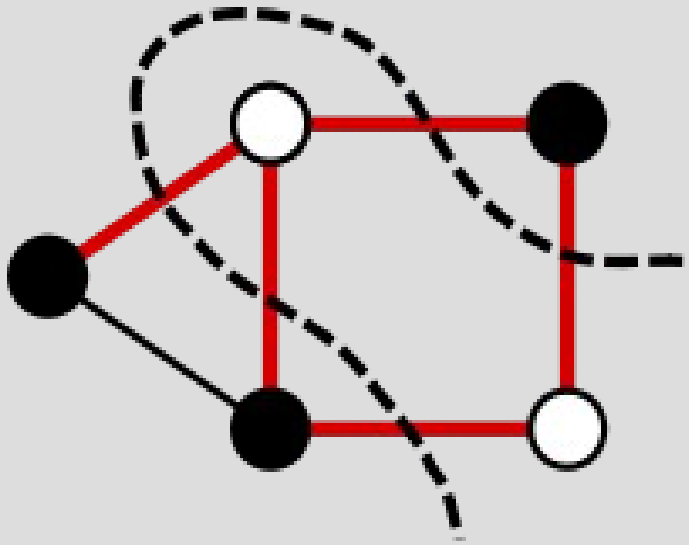
- Régler les paramètres = s'adapter au problème :
  - Définir une solution initiale,
  - Définir une structure de voisinage,
  - Définir les paramètres spécifiques.
- Bon usage : comparer bornes inf. et sup. !
- Critère(s) d'arrêt :
  - Temps limite,
  - Ecart de la valeur de la solution courante à une borne,
  - Combinaison de ces deux critères...

Utiliser la recherche locale pour concevoir des algo. approchés

- En général, la recherche locale ne mène qu'à un optimum local, qui peut alors être arbitrairement moins bon qu'un optimum global...
- Certains problèmes ont la propriété suivante :
  - La valeur de **tout** optimum local est « proche » de celle d'un optimum global,
  - Dans ce cas, la recherche locale est un bon moyen de fournir une bonne solution approchée !
  - Néanmoins, l'analyse nécessite une nouvelle fois d'identifier et d'exploiter de “bonnes” bornes.

Algo. approché par recherche locale pour la coupe maximum (1/7)

- **Contexte** : on cherche à identifier, dans un réseau social, deux « communautés », de façon à maximiser, entre ces communautés, le nombre de paires d'individus qui ne sont pas en relation
- **Formalisation** : on représente le réseau par un graphe, dont les sommets sont les individus, et il y a une arête entre deux sommets si les deux individus correspondants ne sont **pas** en relation. On cherche alors une coupe maximum dans ce graphe.
- **Problème NP-difficile**



Algo. approché par recherche locale pour la coupe maximum (2/7)

- **En résumé :**
  - On souhaite séparer les sommets du graphe en 2 groupes (communautés), de façon à maximiser le nombre d'arêtes entre ces deux groupes de sommets (= **coupe maximum**)
  - Cela revient à minimiser, au sein des communautés, le nombre de paires d'individus qui ne sont pas en relation
  - Cela revient-il également à minimiser, entre les deux communautés, le nombre de paires d'individus qui sont en relation ? Ou bien à maximiser, au sein des communautés, le nombre de paires d'individus qui sont en relation ?

Algo. approché par recherche locale pour la coupe maximum (2/7)

- **En résumé :**
  - On souhaite séparer les sommets du graphe en 2 groupes (communautés), de façon à maximiser le nombre d'arêtes entre ces deux groupes de sommets (= **coupe maximum**)
  - Cela revient à minimiser, au sein des communautés, le nombre de paires d'individus qui ne sont pas en relation
  - Cela revient-il également à minimiser, entre les deux communautés, le nombre de paires d'individus qui sont en relation ? Ou bien à maximiser, au sein des communautés, le nombre de paires d'individus qui sont en relation ?
  - Non : ces deux « mesures » sont différentes !
    - En fait : coupe maximum vs coupe minimum.
    - Exemple : trois arêtes disjointes par les sommets (3 vs 4).

Algo. approché par recherche locale pour la coupe maximum (3/7)

- Soient G (gauche) et D (droite) les 2 groupes
- Voisinage considéré : changer un sommet de côté
- Algorithme par améliorations successives (recherche locale vis-à-vis de ce voisinage) :
  - Initialement, G est vide (D contient tous les sommets)
  - De façon itérative, et tant que cela est possible, changer un sommet de côté (l'ajouter à G s'il se trouve dans D, à D sinon) si la solution obtenue est meilleure

Algo. approché par recherche locale pour la coupe maximum (4/7)

- Dans ce problème, toute solution est admissible :
  - En effet : G quelconque ==> solution admissible si D contient tous les autres sommets
- Borne supérieure sur la valeur optimale ?
  - Borne simple : nombre d'arêtes (noté m)
- Temps d'exécution polynomial ?
  - A chaque itération, on améliore la valeur de la solution d'au moins 1 ==> au plus m itérations

Algo. approché par recherche locale pour la coupe maximum (5/7)

- Analyse du ratio d'approximation ?
  - A la fin de l'algo., au moins la moitié des arêtes incidentes à chaque sommet est dans la coupe
    - Sinon : la solution obtenue en changeant un sommet de côté serait meilleure...
  - Nombre arêtes coupe = somme nombre arêtes coupe incidentes à chaque sommet de G
  - Donc : nombre arêtes coupe ≥ moitié de la somme des degrés des sommets dans G
  - Même chose pour D

Algo. approché par recherche locale pour la coupe maximum (6/7)

- Analyse du ratio d'approximation (suite) ?
  - Ainsi : nombre arêtes coupes ≥ quart de la somme des degrés des sommets du graphe
  - Or : somme des degrés d'un graphe = 2m
  - Donc : nombre arêtes coupe ≥ m/2
  - Valeur optimale au plus m (car borne sup. simple) ==> valeur solution obtenue = au moins **la moitié** de la valeur optimale ==> algorithme 2-approché
  - De nouveau, l'algo. peut être meilleur en pratique (garantie de performance dans le pire cas)

Algo. approché par recherche locale pour la coupe maximum (7/7)

- Ici aussi, la solution obtenue via cette recherche locale peut être un optimum local non global :
  - Exemple : on considère un graphe à 9 sommets.
  - Les m=10 arêtes de ce graphe sont : 1-2, 1-3, 2-4, 3-4, 4-5, 4-6, 5-7, 6-7, 4-8 et 4-9.
  - Solution optimale : G={1,4,7} (de valeur 10)
  - L'optimum local obtenu par recherche locale à partir de la solution initiale G={2,4,5,8} est de valeur 6, et n'est donc pas un optimum global !

A retenir...

- Avantages des algorithmes approchés :
  - Plus rapides (parfois même, beaucoup plus) que les algorithmes exacts pour les problèmes NP-difficiles
  - Possibilité d'exploiter les particularités du problème
  - Parfois, analyse théorique de l'algorithme possible (garantie de performance via ratio d'approximation)
  - Possibilité d'améliorer les solutions approchées à l'aide de la recherche locale ou de métaheuristiques

A retenir (suite)...

- Inconvénients des algorithmes approchés :
  - Certains sont spécifiques à un problème donné
  - Les métaheuristiques, bien que génériques, nécessitent un paramétrage parfois délicat
  - Tous n'offrent pas de garantie de performance a priori
  - Certains peuvent donc, en théorie, fournir une solution arbitrairement moins bonne qu'une solution optimale
  - Plus difficiles à utiliser dans le cas où trouver une solution admissible est déjà un problème ardu