ÉPREUVE COMMUNE DE TIPE 2014 - Partie D

TITRE :

Algorithmes approximatifs

Temps de préparation : ……2 h 15 minutes
Temps de présentation devant les examinateurs : …….10 minutes
Dialogue avec les examinateurs : ……..10 minutes

GUIDE POUR LE CANDIDAT :

Le dossier ci-joint comporte au total : 11 pages
Document principal (11 pages, dont celle-ci) ;

Travail **suggéré** au candidat :

Faire un exposé synthétique, en expliquant le principe des algorithmes approximatifs ainsi que la justification de leur utilité en cas de problèmes difficiles. Pour ce faire, vous pouvez utiliser les exemples proposés dans le dossier, en ajoutant vos propres arguments et éclairer un peu plus les passages qui ne vous semblent pas évidents. Les examinateurs apprécieront, le cas échéant, vos propres exemples dûment justifiés.

Attention : s*i le candidat préfère effectuer un autre travail sur le dossier, il lui est* ***expressément recommandé*** *d'en informer le jury avant de commencer l'exposé.*

CONSEILS GENERAUX POUR LA PREPARATION DE L'EPREUVE :

* Lisez le dossier en entier dans un temps raisonnable.
* Réservez du temps pour préparer l'exposé devant les examinateurs.

- Vous pouvez écrire sur le présent dossier, le surligner, le découper … mais tout sera à remettre aux examinateurs en fin d'oral.
- En fin de préparation, rassemblez et ordonnez soigneusement TOUS les documents (dossier, transparents, *etc.*) dont vous comptez vous servir pendant l'oral. En entrant dans la salle d'oral, vous devez être prêt à débuter votre exposé.
- A la fin de l'oral, vous devez remettre aux examinateurs le présent dossier dans son intégralité. Tout ce que vous aurez présenté aux examinateurs pourra être retenu en vue de sa destruction.

IL EST INTERDIT DE SORTIR LE DOSSIER DU SITE DE L'EPREUVE

Algorithmes approximatifs

1. Introduction.

La *combinatoire* est la branche des mathématiques qui étudie les structures discrètes, composées d'un nombre fini d'éléments. Les questions classiques concernent l'existence ou le dénombrement : "*étant donné un certain ensemble, existe-t-il des éléments de cet ensemble satisfaisant des propriétés ou contraintes données*?" et le cas échéant "*combien y en a-t-il* ?". Une autre question plus récente est de savoir "*quelle est la meilleure structure pour un critère donné* ?".

Trivialement, une application quelconque d'un ensemble fini E dans $\mathbb{R}$ admet son maximum (minimum) et pour cette raison très souvent les théoriciens n'attachaient pas beaucoup d'importance à la combinatoire.

De manière générale, le schéma suivant est typique des problèmes d'optimisation combinatoire : on se donne un ensemble fini E et on s'intéresse à une partie de $\mathscr{P}(E)$, famille des sous-ensembles de E. Si E compte n éléments, $\mathscr{P}(E)$ en a 2^n. Il est clair que la procédure triviale de recherche d'une solution par énumération, théoriquement possible, serait d'une efficacité désastreuse.

La distinction entre l'existence d'objets mathématiques et leur construction est relativement récente. Le souci de prendre en considération le temps d'exécution des algorithmes n'apparaît que dans les années 1950–1960, dans plusieurs travaux, dont ceux de John von Neumann bien après les travaux d'Alan Turing, le concepteur du premier modèle mathématique de l'ordinateur. L'idée de la caractérisation des algorithmes efficaces remonte à Jack Edmonds (1965).

Si l'on perçoit assez bien ce que ***fait*** un algorithme, on a plus de difficulté à voir ce qu'il ***est***. Dans les années 30, le mathématicien Alan Turing parvient à formaliser l'idée intuitive qu'un algorithme est une procédure que l'on peut exécuter mécaniquement, sans réfléchir. Il modélisa un processus de calcul par une machine, simple mais précise, composée d'un ruban infini divisé en cases et d'une tête capable d'effectuer seulement quatre actions élémentaires : quand la tête accède à une case elle peut : *lire* le contenu, *écrire* un caractère en effaçant l'ancien, *se déplacer à gauche*, *se déplacer à droite*. Ainsi on peut dédier une machine à toute procédure décomposable en une séquence des dites actions. La réalisation effective d'une telle machine n'est pas essentielle : une machine de Turing est une formulation mathématique d'un algorithme.

Pour ne pas alourdir le présent dossier, nous proposons de remplacer la définition rigoureuse d'un algorithme, par une autre, suffisamment précise pour le propos que nous voulons développer :

DEFINITION 1 : *Un **algorithme** de résolution d'un problème P donné est une procédure, décomposable en opérations élémentaires, transformant une chaîne de caractères représentant les **données** de n'importe quel exemple (I) du problème P en une chaîne de caractères représentant les **résultats** de (I).*

Ainsi, nous n'attendons pas d'un algorithme qu'il résolve le problème P général en termes de paramètres littéraux mais seulement, chaque fois que les données du problème ont été fixées (l'instance I), qu'il soit capable de donner la solution du cas correspondant.

Pour mesurer l'efficacité d'un algorithme donné, nous allons tenter d'établir une relation entre la durée d'exécution dudit algorithme, exprimée en nombre d'opération élémentaires, et la taille de l'exemple traité, exprimée en nombre de caractères nécessaires pour coder les données de l'exemple traité.

On dit qu'un algorithme est ***polynomial*** si son comportement se décrit de la façon suivante : *pour des données qui n'occupent pas plus de* $\mathbf{n}$ *octets de mémoire d'un ordinateur, l'algorithme nécessite moins de* $c\mathbf{n}^k$ *opérations élémentaires (c et k sont des constantes).*

Cette définition impose seulement la manière dont le temps d'exécution augmente en fonction des données. Le terme algorithme polynomial vient du fait que la fonction $c\mathbf{n}^k$ est un polynôme en $\boldsymbol{n}$.

Un algorithme polynomial est dit ***efficace***. Le problème pouvant être résolu par un tel algorithme est dit ***facile***.

Ce choix de la fonction polynomiale était fait principalement pour deux raisons :

1° la fonction polynomiale croît moins vite que toute fonction exponentielle (c'est-à-dire que si $a > 1$ on ne peut pas trouver b tel que $a^{\boldsymbol{n}} < \boldsymbol{n}^b$) ;

2° la classe de polynômes est fermée pour la composition : un polynôme d'un polynôme est un polynôme.

L'ensemble des problèmes faciles forme une classe notée P (comme polynomiale).

Néanmoins, la "définition" proposée de l'algorithme polynomial pose deux problèmes :

a) Quelles sont les opérations élémentaires qu'il faut prendre en compte ?

b) Quel codage des données permet de mesurer la taille de l'exemple traité ?

Ad a) : on peut comptabiliser les quatre opérations élémentaires, les comparaisons, etc., mais compte tenu de la propriété 2° ci-dessus on peut dénombrer les opérations de base de l'ordinateur (affectation à un registre, opérations sur les bits d'un mot …) car, dans cette énumération, une opération élémentaire sera remplacée par une fonction polynomiale du nombre de bits nécessaire pour coder les données. Cependant ce sont les opérations effectuées qu'il faut comptabiliser et non les instructions d'un langage algorithmique. Dans les langages évolués il est parfois difficile de dénombrer les opérations effectuées par un algorithme récursif.

Ad b) : un codage des données sera *raisonnable* si les nombres ne sont pas codés sous forme unaire[1]. Pour stocker le nombre entier N il suffit de $\lceil \log_2(N+1) \rceil$ bits si on utilise le codage binaire[2] (le symbole $\lceil x \rceil$ désigne le plus petit entier supérieur ou égal à x). Ainsi on considère qu'un entier possède d'une part sa valeur ($= N$) et d'autre part sa taille ($= \lceil \log_2(N+1) \rceil$). Même en utilisant le formalisme de la machine de Turing on ne sait pas donner de définition plus précise du codage raisonnable.

Les trois exemples suivants d'addition de deux entiers vont illustrer l'importance du codage choisi. Les données sont entrées sous la forme : A ; B. Pour coder, on va utiliser 0, 1 et un séparateur " ; " en supposant que chacun de ces symboles occupe une case mémoire.

EXEMPLE 1: Codage binaire, algorithme usuel (l'addition bit par bit) : la taille des données est $n = \lceil \log_2(A+1) \rceil + \lceil \log_2(B+1) \rceil + 1$; le nombre d'opération est borné par $3n$: une addition par bit ou deux, s'il y une retenue, et une comparaison de la somme à 1; cet algorithme est donc polynomial.

[1] Le codage unaire représente tout entier naturel n par n occurrences de 1 suivies par un 0 faisant office de délimiteur.

[2] Dans un système de numération utilisant la base 2, chaque entier naturel n est représenté par une séquence de bits (« *binary digits* ») c'est-à-dire une séquence de chiffres binaires 0 ou 1, cela correspond au codage binaire de n.

EXEMPLE 2 : Codage binaire, algorithme "*compter sur ses doigts*" :
la taille de données est $n = \lceil \log_2(A+1) \rceil + \lceil \log_2(B+1) \rceil + 1$ mais l'algorithme proposé est :

tant que A > 0 **faire**
    A:=A–1; B:=B+1
fin tant que

A la fin de l'algorithme la somme cherchée vaut B. Le nombre d'opération $f(n)$ = A et, dans le plus mauvais cas, si $A = 2^k$ et B = 0, cet algorithme n'est pas polynomial.

EXEMPLE 3 : Codage unaire, algorithme "*compter sur ses doigts*" :

la taille des données est n = A+B+1; le nombre d'opérations est borné par n et notre algorithme, précédemment exponentiel, devient polynomial suite au codage des données non raisonnable.

2. Problèmes faciles.

Les problèmes faciles existent. En voici un exemple :

EXEMPLE 4: On dispose de n tuyaux avec les longueurs $b_1, b_2, \ldots, b_n$ en plastique bleu pour l'eau froide et de n tuyaux avec les longueurs $r_1, r_2, \ldots, r_n$ en plastique rouge pour l'eau chaude. On doit les vendre par paire (rouge, bleu) mais il faut ajuster les longueurs, c'est-à-dire raccourcir le tuyau plus long. Le bout coupé sera perdu. Comment coupler (r_i, b_j) pour minimiser la longueur totale des bouts coupés (longueur des chutes) ?

Ce problème est facile car il existe un algorithme efficace pour construire une solution optimale. En supposant que $b_1 \le b_2 \le \cdots \le b_n$ et $r_1 \le r_2 \le \cdots \le r_n$ nous allons prouver que le couplage $\{(1;1), (2;2), \ldots, (n;n)\}$ est optimal, c'est-à-dire que :

$$|b_1 - r_1| + |b_2 - r_2| + \cdots + |b_n - r_n| = \min_\pi \{|b_1 - r_{\pi(1)}| + |b_2 - r_{\pi(2)}| + \cdots + |b_n - r_{\pi(n)}|\},$$

où π est une permutation de $(1, 2, \ldots, n)$. Il est facile de vérifier que l'énoncé est vrai pour $n = 2$. En effet, notons $\alpha = \max\{b_1; r_1\}$ et $\beta = \min\{b_2; r_2\}$; si $\alpha \ge \beta$, alors

$$|b_1 - r_1| + |b_2 - r_2| = |b_1 - r_2| + |b_2 - r_1|,$$

sinon

$$|b_1 - r_1| + |b_2 - r_2| = |b_1 - r_2| + |b_2 - r_1| - 2(\beta - \alpha) < |b_1 - r_2| + |b_2 - r_1|.$$

Quand $n \ge 2$ et le couplage est différent de $\{(1;1), (2;2), \ldots, (n;n)\}$ alors il existe deux couples *inversés*, c'est-à-dire deux couples $(i ; j)$ et $(k ; l)$ avec $i < k$ et $j > l$. Le nouveau couplage où les couples $(i ; j)$ et $(k ; l)$ ont étaient remplacés par $(k ; j)$ et $(i ; l)$ aura la longueur totale de chutes inférieure ou égale à la précédente, comme le montre le raisonnement pour $n = 2$. En répétant cette procédure tant qu'il existe des couples inversés, on voit que le couplage final $\{(1;1), (2;2), \ldots, (n;n)\}$ a la longueur totale de chutes inférieure ou égale à celle du couplage initial.

L'algorithme précédent incite à définir une famille d'algorithmes appelés ***gloutons***. Ce nom a été proposé par J. Edmonds en 1967 (*greedy* – en anglais), car il consiste à "manger" les éléments de l'ensemble considéré dans un certain ordre sans jamais remettre en question un choix déjà effectué (ce qui est mangé est mangé).

Supposons $E = \{e_1, e_2, \ldots, e_n\}$ et que la famille $S \subset \mathcal{P}(E)$ de solutions admissibles est ***héréditaire***, c'est-à-dire si $A' \in S$ et $A' \subset A$ alors $A \in S$.

Le principe de l'algorithme glouton peut être présenté de la manière suivante :

ALGORITHME GLOUTON :

$X \coloneqq \varnothing$
pour $i \coloneqq 1$ ***à*** n ***faire***
    si $X \cup \{e_i\} \in S$ ***alors*** $X \coloneqq X \cup \{e_i\}$
    fin si
fin pour
fin algorithme

Pour qu'un algorithme glouton soit efficace il faut que le test ($X \cup \{e_i\} \in S$?) puisse être réalisé simplement et il faut que l'ordre de parcours des éléments de E soit tel que l'algorithme glouton donne une solution optimale, ce qui n'est pas toujours le cas.

3. Problèmes difficiles.

Pour certains problèmes, bien connus et avec des applications industrielles très importantes, nous ne savons pas démontrer qu'ils sont faciles. En voici trois exemples :

EXEMPLE 5: le problème du **VOYAGEUR DE COMMERCE.** Un voyageur de commerce ayant n villes à visiter souhaite établir une tournée qui lui permette de passer une seule fois dans chaque ville pour finalement revenir à son point de départ, ceci en minimisant la longueur du chemin parcouru.

On voit aisément que beaucoup de problèmes quotidiens sont équivalents à ce problème, par exemple les tournées du facteur ou le ramassage des ordures ménagères.

EXEMPLE 6 : le problème du **SAC A DOS**. On dispose de n objets ayant chacun un poids p_i et une valeur v_i (i = 1, 2,…, n). Un randonneur doit effectuer une sélection (déterminer un sous-ensemble de ces objets) dont le poids total soit inférieur ou égal à une valeur donnée (le poids maximum qu'il peut supporter) et dont la somme des valeurs soit maximum. Ce problème doit son nom au scénario qui est souvent utilisé pour l'introduire, mais, en réalité, il est assez mal adapté à la préparation des randonnées, car il ne tient pas compte des utilités croisées, c'est-à-dire du fait que le dentifrice sans la brosse-à-dents et la boîte de sardines sans l'ouvre-boîte sont de faible utilité. Tout ceci n'empêche pas ce problème d'avoir de nombreuses applications pratiques.

En associant à chaque objet i (i =1, 2,…, n) une variable qui prend la valeur 1 si le $i^{\text{ème}}$ objet fait partie de la sélection et 0 sinon. Le problème du sac à dos se formule ainsi :

Maximiser : $v_1 x_1 + v_2 x_2 + \cdots + v_n x_n$
sous : $p_1 x_1 + p_2 x_2 + \cdots + p_n x_n \leq b$
$x_i \in \{0, 1\}$ i = 1, 2,…, n.

EXEMPLE 7: le problème de la « **MISE EN BOITE** ». Trouver le rangement le plus économique possible pour un ensemble d'articles dans des boîtes. Dans le problème classique, les données sont : un nombre infini de boîtes de taille 1 et une liste 1, 2, …, n d'articles i de taille a_i (correspondant à une fraction de la taille d'une boîte). On cherche à trouver le rangement valide (la somme des tailles des articles affectés à une boîte doit être inférieure ou égale à 1) pour tous ces articles qui minimise le nombre de boîtes utilisées.

Essayons par exemple, d'appliquer l'algorithme glouton pour le problème du sac à dos suivant :

$$\begin{aligned} \text{Maximiser :} \quad & 10x_1 + 8x_2 + 5x_3 \\ \textit{sous :} \quad & 6x_1 + 5x_2 + 4x_3 \leq 9 \\ & x_i \in \{0, 1\}\ i = 1, 2, 3. \end{aligned}$$

Supposons un ordre raisonnable – donnons la priorité aux objets suivant la valeur du ratio $^{\text{utilité}}/_{\text{poids}} := v_i/p_i$. On obtient la solution pas fameuse : $x_1 = 1$, $x_2 = x_3 = 0$ de valeur 10 pendant que la solution optimale a la valeur 13. Ainsi on voit bien qu'un algorithme glouton est une approche simpliste qui est loin de donner toujours une solution optimale.

Mais de quels arguments pouvons-nous disposer pour affirmer qu'un problème est ***difficile*** ? Le fait qu'on ne connaisse pas aujourd'hui d'algorithme efficace pour résoudre ce problème provient-il de notre inaptitude à en découvrir un ou signifie-t-il qu'un tel algorithme ne peut exister car le problème est intrinsèquement complexe ?

Pour comparer les difficultés des différents problèmes combinatoires, on peut essayer de transformer un problème en un autre. On dit qu'on peut transformer le problème d'optimisation *P* en un autre problème d'optimisation *P'*, s'il existe un algorithme polynomial qui transforme les données de chaque instance (exemple) *I* de *P* en données d'une instance (exemple) *I'* de *P'* de façon telle que l'on puisse toujours déduire facilement la solution optimale de *I* si on connaît la solution optimale de l'instance transformée *I'*. Il n'est pas nécessaire d'atteindre toutes les instances de *P'*; en revanche, il est indispensable de transformer toutes les instances de *P* en instances de *P'* et ceci sans connaître la solution optimale. Il faut souligner qu'on s'intéresse seulement aux transformations efficaces, c'est-à-dire où la taille des données du problème transformé est bornée par une fonction polynomiale de la taille des données du problème initial. Si cette transformation existe on va dire que le problème *P* se *réduit polynomialement* au problème *P'*. En conséquence, s'il existe un algorithme polynomial pour résoudre le problème *P'*, alors il existe un algorithme polynomial pour résoudre le problème *P*. En attendant la découverte éventuelle de cet algorithme polynomial pour *P'*, l'existence d'une réduction polynomiale de *P* en *P'*, nous permet de dire que le problème *P'* est *au moins aussi difficile* que *P*.

Pour illustrer le concept de réduction sur des exemples, nous allons utiliser les termes de la théorie des graphes. Un *graphe G* est un couple (X, E) où $X = \{x_1, x_2, ..., x_n\}$ est un ensemble fini non vide d'éléments appelés *sommets* de *G* (parfois *nœuds* ou *points*) et $E = \{e_1, e_2, ..., e_m\}$ est une famille de paires (non ordonnées) d'éléments de *X*, appelés *arêtes* de *G*. Pour une arête $e = \{x, y\}$, *x* et *y* sont ses *extrémités*. Si l'arête $\{x, y\}$ existe on dit que les *sommets x* et *y* sont *adjacents*. Un sommet $x \in X$ et une arête $e \in E$ sont *incidents* si *x* est une extrémité de *e*. Le *degré* $d(x)$ d'un sommet *x* est le nombre d'arêtes incidentes à *x*.

On dit que le graphe $H = (Y, F)$ est un *sous-graphe* du graphe $G = (X, E)$ *induit par Y* lorsque $Y \subseteq X$ et *F* est formé de la totalité des arêtes de *E* ayant leurs deux extrémités dans *Y*. On note $G - x$ le sous-graphe induit par $X - \{x\}$.

Une *chaîne de longueur k* est une séquence alternée de sommets et d'arêtes $\Gamma = (x_0, e_1, x_1, e_2, ..., x_{k-1}, e_k, x_k)$ telle que, pour tout *i*, les extrémités de e_i sont x_{i-1} et x_i.

Etant donnés un graphe $G = (X ; E)$ et deux sommets particuliers a et b, appelons $\boldsymbol{P_1}$ le problème de trouver la plus courte chaîne reliant a et b (c'est-à-dire une chaîne qui commence en a et finie en b, et pour laquelle la longueur k est la plus petite possible).

On obtient un graphe orienté $G = (X ; A)$ lorsque, pour toute arête $e \in E$ d'un graphe non orienté $G = (X ; E)$ ses extrémités sont ordonnées. Chaque arête se transforme en un *arc* qui est un couple de sommets (initial et terminal). Si $a = (x, y)$ est un arc, on dit que "*y est un successeur de x*" et "*x est un prédécesseur de y*".

Un *chemin* de longueur k est une séquence alternée de sommets et d'arcs $\Gamma = (x_0, e_1, x_1, e_2, ..., x_{k-1}, e_k, x_k)$ telle que, pour tout i, e_i admet x_{i-1} comme sommet initial et x_i comme extrémité terminale.

Etant donnés un graphe orienté $G = (X ; A)$ et deux sommets particuliers a et b, appelons $\boldsymbol{P_2}$ le problème de trouver le plus court chemin de a à b.

Sans résoudre les problèmes $\boldsymbol{P_1}$ et $\boldsymbol{P_2}$ nous allons réduire le problème $\boldsymbol{P_1}$ au $\boldsymbol{P_2}$: à chaque graphe non orienté $G = (X ; E)$ on va associer un graphe orienté $G = (X ; A)$ en remplaçant chaque arête $e = \{x, y\}$ par deux arcs opposés $a_1 = (x, y)$ et $a_2 = (y, x)$. Cette transformation est polynomiale car le nouveau graphe a le même nombre de sommets et le nombre d'arcs est le double du nombre d'arêtes. Ainsi chaque graphe non orienté est transformé en un graphe orienté de façon telle que :

– si, dans le graphe original (non orienté), il existe une chaîne de longueur k reliant a et b alors il existe un chemin de longueur k de a à b dans le graphe transformé (orienté) ;

– s'il existe un chemin de longueur k de a à b dans le graphe transformé alors il existe une chaîne reliant a et b dans le graphe original (non orienté).

La transformation proposée, permet d'identifier facilement une plus courte chaîne dans le graphe original (non orienté) quand on connait le plus court chemin dans le graphe transformé.

S'il existe un algorithme polynomial pour résoudre $\boldsymbol{P_2}$, nous avons, grâce à la réduction présentée ci-dessus, un algorithme polynomial pour $\boldsymbol{P_1}$. Nous dirons que $\boldsymbol{P_2}$ est *au moins aussi difficile* que $\boldsymbol{P_1}$.

Voici un autre exemple. On appelle *stable* un ensemble de sommets deux à deux non adjacents (voir la figure 1).

Le nombre maximum de sommets d'un stable désigné par [*alpha*] $\alpha(G)$ s'appelle *nombre de stabilité*.

Appelons **STABLEMAX** le problème d'optimisation combinatoire qui consiste à déterminer le nombre de stabilité d'un graphe donné (c'est-à-dire déterminer la plus grande taille d'un stable).

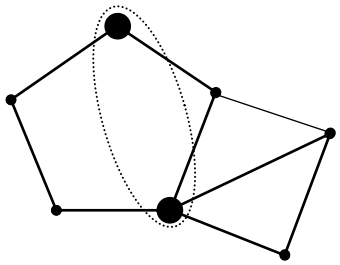

Fig.1 : Un *stable* d'un graphe G (ici $\alpha(G) = 3$).

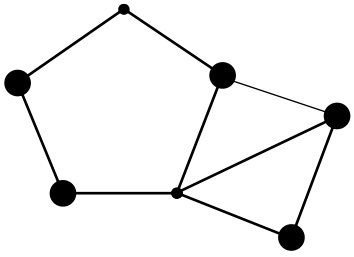

Fig.2 : Un *transversal* d'un graphe G (ici $\tau(G) = 4$).

On dénote par $T \subseteq X$ un ensemble *transversal* c'est-à-dire un ensemble de sommets tel que toute arête de G ait au moins une de ses extrémités dans T (voir la figure 2). Le nombre minimum de sommets d'un transversal désigné par [tau] $\tau(G)$ s'appelle le nombre de transversalité. Appelons **TRANVERSALMIN** le problème d'optimisation combinatoire qui consiste à déterminer le nombre de transversalité d'un graphe donné (c'est-à-dire déterminer la plus petite taille d'un transversal).

On peut déduire directement des définitions les propriétés suivantes :

PROPRIETE 1: Si $T \subset X$ est un transversal, alors $X \setminus T$ est un stable.

La réciproque est vraie aussi :

PROPRIETE 2: Si $S \subset X$ est un stable, alors $X \setminus S$ est un transversal.

CONCLUSION : $\alpha(G) + \tau(G) = |X|$.

Grace à ces propriétés on voit clairement qu'on peut réduire **STABLEMAX** en **TRANVERSALMIN**. Si un transversal de cardinalité minimum est trouvé (même difficilement), nous pouvons en déduire rapidement un stable de cardinalité maximum. Cela veut dire que s'il existe un algorithme polynomial pour résoudre **TRANVERSALMIN**, il permet en même temps de résoudre **STABLEMAX**. Il n'y a pas vraiment de transformation : les données restent les mêmes (le même graphe) mais cela n'est pas nécessaire – on peut se permettre d'augmenter raisonnablement (polynomialement) la taille de données transformées par rapport à la taille du problème original. Ainsi on peut dire que **TRANVERSALMIN** est *au moins aussi difficile* que **STABLEMAX**.

On constate aisément que **TRANVERSALMIN** se transforme en **STABLEMAX** : si un stable de cardinalité maximum est trouvé (même difficilement), nous pouvons en déduire rapidement un transversal de cardinalité minimum. En plus, s'il existe un algorithme polynomial pour résoudre **STABLEMAX**, le même algorithme, avec notre "déduction", permet de résoudre **TRANVERSALMIN**.

Cet exemple montre que parfois deux problèmes avec les formulations a priori différentes peuvent avoir la même difficulté. En général, l'idée de la réduction polynomiale permet d'introduire dans l'ensemble des problèmes combinatoires une relation d'ordre : *être au moins aussi difficile que*.

On pourrait s'attendre à ce que la difficulté ainsi définie augmente sans cesse, mais ce n'est pas le cas, car la difficulté atteint un maximum. A notre surprise il existe une famille de problèmes les plus difficiles, appelés NP-*difficiles* (le sigle NP vient de "*polynomial non-déterministe*" – un formalisme mathématique pour caractériser plus précisément les problèmes dont on veut étudier la difficulté). Les cinq problèmes cités ci-dessus : le problème du **VOYAGEUR DE COMMERCE**, du **SAC A DOS**, de la « **MISE EN BOITES** », **STABLEMAX** et **TRANVERSALMIN** appartiennent à cette famille. Une grande partie des problèmes appliqués se retrouve dans la famille des problèmes NP-*difficiles* – plus de mille sont connus à ce jour. Ces problèmes sont aussi appelés *universels* car la découverte de l'algorithme polynomial pour l'un d'eux impliquerait automatiquement l'existence des algorithmes polynomiaux pour tous les autres.

4. Conséquences de la NP-*difficulté* d'un problème.

Le chercheur ou l'ingénieur confronté à un nouveau problème d'optimisation combinatoire va d'abord essayer de découvrir un algorithme polynomial. En cas d'insuccès il va tenter de montrer que le problème considéré est NP-*difficile* pour éviter de perdre du temps à rechercher un algorithme polynomial.

Si cette propriété est prouvée on sait qu'au cas où les exemples à traiter sont de trop grande taille, on est autorisé à faire appel aux algorithmes heuristiques. On préfère avoir rapidement une mauvaise solution que pas de solution du tout. Ensuite on peut tenter d'améliorer la solution obtenue avec des procédures supplémentaires, simples et rapides.

Un algorithme est dit *heuristique* (ou *approximatif*) s'il conduit toujours à une solution réalisable (c'est-à-dire qui vérifie les contraintes du problème) mais pas nécessairement à une solution optimale.

Un algorithme approximatif est une procédure très précisément définie (pas approximativement du tout). Ce qui est *approché* c'est la solution trouvée par cet algorithme.

Les méthodes approximatives sont très souvent spécifiques des problèmes qu'elles se proposent de résoudre.

Traditionnellement, l'inventeur d'un algorithme approximatif se contentait d'illustrer les performances de son heuristique sur quelques exemples. Cette manière de faire présente des inconvénients évidents. Si l'algorithme est utilisé avec un jeu de données différent, on n'a aucune idée de la qualité de la solution qui pourra en sortir. On pourrait essayer d'examiner cet algorithme sur une batterie de tests tirés au hasard, mais on constate qu'avec les données aléatoires cet algorithme est presque toujours très performant. Et sur un plan théorique, on n'a aucune indication sur les raisons pour lesquelles les performances risqueraient de se dégrader dans certaines conditions. Ainsi, pour évaluer des algorithmes approximatifs, le critère du plus mauvais cas est le mieux adapté.

5. Algorithmes approximatifs avec performance garantie.

Pour caractériser la performance d'un algorithme approximatif sur l'exemple I d'un problème P nous pouvons utiliser l'erreur relative $R(I) = |\,\mathrm{AP}(I) - \mathrm{OPT}(I)\,| \,/\, \mathrm{OPT}(I)$ en cas de minimisation et $R(I) = |\mathrm{OPT}(I) - \mathrm{AP}(I)| \,/\, \mathrm{AP}(I)$ en cas de maximisation où $\mathrm{AP}(I)$ désigne la valeur de la solution obtenue en appliquant notre algorithme approximatif et $\mathrm{OPT}(I)$ la valeur optimale (que l'on suppose positive sans perte de généralité).

S'il existe $\varepsilon \geq 0$ tel que $R(I) \leq \varepsilon$ pour tous les exemples I du problème P, alors on dit que l'algorithme est une *ε-approximation*. Un algorithme est une *f*(*n*)-*approximation* quand $R(I) \leq f(n)$ pour tous les exemples I du problème P, où $f(n)$ est une mesure de la taille des données de l'exemple I.

Essayons l'algorithme glouton suivant pour résoudre **TRANSVERSALMIN** :

Données : graphe $G = (X, E)$;
Résultat : un transversal.

algorithme1:
  $T \coloneqq \varnothing$;
  tant que $E \neq \varnothing$ **faire**
    choisir un sommet $x \in X$ avec le plus grand degré;
    $G \coloneqq G - x$ et $T \coloneqq T \cup \{x\}$.
  fin tant que
fin algorithme1.

Cet algorithme est polynomial, car le nombre d'opérations est borné par n – le nombre de sommets du graphe donné. Evidemment, il va falloir perdre du temps pour ordonner les sommets suivant les degrés décroissants ($n\log_2(n)$). Cet algorithme choisit successivement les sommets qui sont incidents au maximum d'arêtes alors, intuitivement, il devrait fournir une bonne solution (même quand elle n'est pas optimale). Cependant, on verra bien sur une famille d'exemples spécialement choisis que les performances de l'algorithme 1 ne sont pas fameuses.

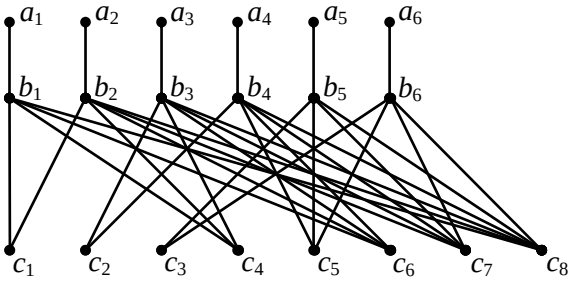

Fig.3 : Exemple de contreperformance de l'algorithme1.

L'application de cet algorithme au graphe de la figure 3 nous amène à choisir dans l'ordre les sommets $c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1$ et pour obtenir un transversal il faudra encore choisir un sommet de chaque arête $\{a_i, b_i\}$ pour i =1, 2, …, 6. On a ainsi trouvé un transversal comportant 14 sommets alors que les 6 sommets b_i suffisent. L'erreur relative est de 8/6 ≈ 133,33%. Pire encore, cet exemple peut être étendu et conduire l'algorithme1 à des erreurs arbitrairement grandes. En effet, remplaçons les 6 arêtes de l'exemple précédent par les n arêtes $\{a_i, b_i\}$ (i =1, 2, …, n). Ajoutons ensuite les $\lfloor n/2 \rfloor$ sommets c_j (le symbole $\lfloor \cdot \rfloor$ désigne la partie entière) reliés aux deux sommets b_{2j-1} et b_{2j} (le sommet b_n ne sera pas utilisé si le nombre n est impair), puis les $\lfloor n/3 \rfloor$ nouveaux sommets c_i reliés aux trois sommets des triplets successifs et ainsi de suite avec les quadruplets etc., jusqu'au dernier sommet c_L relié aux $n-1$ sommets b_i. Pour chaque n, l'algorithme1 conduit à un transversal comportant $L(n) + n$ sommets où $L(n) := \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \cdots + \lfloor n/n-1 \rfloor$ est le nombre des sommets c ajoutés. L'erreur relative $L(n)/n$ croit aussi vite que $ln(n)$. On peut démontrer que l'algorithme proposé est une $ln(n)$-*approximation*.

On serait plutôt intéressé par une heuristique avec l'erreur relative bornée.

Pour résoudre **TRANSVERSALMIN** considérons alors une autre heuristique gloutonne :

Données : graphe $G = (X, E)$;
Résultat : un transversal.

algorithme2:
  $T := \varnothing$;
  tant que $E \neq \varnothing$ **faire**
    choisir une arête $e = \{x, y\} \in E$;
    $G := G - x - y$ et $T := T \cup \{x\} \cup \{y\}$.
  fin tant que
fin algorithme2.

Cet algorithme ne paraît pas a priori très bon : pourquoi mettre les deux extrémités de e dans T alors qu'une seule suffirait ? Cependant les différentes arêtes choisies par l'algorithme sont deux à deux non adjacentes. Il s'ensuit qu'un transversal de taille minimum contiendra au moins autant de

sommets que l'algorithme a appelé d'arêtes c'est-à-dire ½|T|. Ainsi OPT(I) ≥ ½AP(I) ; ce qui implique $R(I) \leq 1$, qui signifie que l'algorithme 2 est une 1-*approximation*.

Il peut se faire qu'un algorithme A dont on est intuitivement persuadé qu'il est presque toujours meilleur que l'algorithme B puisse conduire, sur des exemples spécialement conçus, à des performances désastreuses tandis que l'algorithme B a une performance garantie.

6. Conclusions.

Actuellement les algorithmes approximatifs polynomiaux avec des performances garanties pour les problèmes NP-difficiles font l'objet de nombreuses recherches.

En ce qui concerne le VOYAGEUR DE COMMERCE on a trouvé d'abord une 1-*approximation*. En 1976 N. Christofides a proposé une 0,5-*approximation*. Il aura fallu attendre 36 ans (A. Sebo, J. Vygen 2012) pour obtenir une 0,4-*approximation*. Les améliorations sont difficiles à obtenir, car il a était démontré que l'existence d'un algorithme approximatif polynomial avec une "trop bonne" performance garantie pour un problème NP-difficile impliquerait l'existence d'un algorithme polynomial exact.

Pour le problème de la « MISE EN BOITES » on connaît une $^2/_9$-*approximation*.

Pour le SAC A DOS :

$$\begin{aligned} &\text{Maximiser :} && v_1 x_1 + v_2 x_2 + \cdots + v_n x_n \\ &\textit{sous :} && p_1 x_1 + p_2 x_2 + \cdots + p_n x_n \leq b \\ & && x_i \in \{0, 1\}\ i = 1, 2, \ldots, n \end{aligned}$$

il existe même un algorithme approximatif avec une précision quelconque garantie. Plus précisément, $\forall \varepsilon > 0$ on peut trouver une solution ($\underline{x}_1$; $\underline{x}_2$; … ; $\underline{x}_n$) avec $\underline{x}_i \in \{0;1\}$ i=1, 2, …, n et $p_1\underline{x}_1 + p_2\underline{x}_2 + \cdots + p_n\underline{x}_n \leq b$ telle que :

$$v_1\underline{x}_1 + \cdots + v_n\,\underline{x}_n \geq (1-\varepsilon)\max\{v_1x_1 + \cdots + v_nx_n \mid p_1x_1 + \cdots + p_nx_n \leq b,\ x_i \in \{0;1\}\ i = 1, 2, \ldots, n\}.$$

L'inconvénient est que cet algorithme est polynomial mais par rapport à la valeur $^1/_\varepsilon$. Nous savons déjà que les données de taille k (k cases de mémoire avec les 1), représentent la valeur 2^k–1. Ainsi cet algorithme est exponentiel (non polynomial) par rapport à la taille des données. La NP-difficulté persiste !