CPA-2016 - Cours 6
Arbre de Steiner (suite) – Complexité des problèmes – Recherche locale

M. Soria

1 Arbre de Steiner

1.1 Programmation Dynamique

1.1.1 Algo de base pour le coût

Algo qui renvoie le coût de l'arbre de Steiner (mais pas l'arbre)
Récurrence : $G =< S, A, w >$ graphe valué, avec $|S| = n$. Soit $V \subset S$ ensemble des sommets terminaux $|V| = k$. Récurrence sur la taille de V pour le calcul (du coût) de l'arbre de Steiner.
– Si $|V| = 2$, cela revient à calculer le PPC entre les 2 points dans G
– Sinon supposons qu'on sait calculer Steiner pour tout ensemble de taille $\leq i$; soit $U \subset V$ de taille i, et $x \in V \setminus U$. Supposons d'abord que x est une feuille. Alors dans Steiner($U \cup x$), x est relié par un chemin à un sommet y qui est tel que soit $y \in U$, soit $y \notin U$ et y de degré au moins 2.
Dans le premier cas, en posant $U' := y$, on a bien

$$p(U \cup x) = min_{y \in V, \emptyset \subset U' \subset U}(PPC(x, y) + p(U' \cup y) + p(U \setminus U' \cup y))$$

où $p(U)$ représente le coût min d'un arbre de Steiner couvrant U. Pour le second cas, l'équation est vraie car y est le point de rencontre d'une chaîne partant de x, d'un arbre couvrant $U' \cup y$ et d'un arbre couvrant $U \setminus U' \cup y$. Enfin, si x n'est pas une feuille, l'équation est vraie pour $y = x$.

Algorithm 1 Fonction CoutSteiner($G =< S, A, w >, V$)

```
for all (x, y) ∈ S × S do
  PPC(x, y) ← PCC(x, y, G)
end for
T := maxint
for i := 2 to k − 1 do
  for all U ⊂ V s.t. |U| = i and x ∈ V − U do
    for all y ∈ S and U = U' ∪ U'', U' ≠ ∅, U'' ≠ ∅, U' ∩ U" = ∅ do
      T(U ∪ x) = min_{y,U',U''}(PPC(x, y) + T(U' ∪ y) + T(U'' ∪ y))
    end for
  end for
end for
return T
```

Complexité : 1) la recherche des PPC en $O(n^3)$.
2) Ensuite pour chaque i, une fois choisi U et x ($\binom{k}{i+1}$ choix possibles), partitionner U (2^i façons de faire) et pour tout y (n choix possibles), calculer la formule (on a deja calculé tous les $T(U)$ pour $|U| < i$).
D'ou complexité de l'ordre de $\sum_{i=1}^{k-1} \binom{k}{i+1} 2^i n = O(n3^k)$.

1.1.2 Construction des arêtes de l'arbre

Garder la trace des ensembles et des sommets qui sont atteints dans le calcul des $T(U \cup x)$.

2 Complexité des Problèmes

Un problème P est dit *polynomial* ou dans la classe **P** s'il existe un algorithme polynomial (en la taille des données) qui le résout.

Un rappel sur complexité et temps d'exécution : voir Tables 1 et 2.

Fonction de complexité	Taille n					
	10	20	30	40	50	60
n	$0,01$ µs	$0,02$ µs	$0,03$ µs	$0,04$ µs	$0,05$ µs	$0,06$ µs
n^2	$0,1$ µs	$0,4$ µs	$0,9$ µs	$1,6$ µs	$2,5$ µs	$3,6$ µs
n^3	1 µs	8 µs	27 µs	64 µs	125 µs	216 µs
n^5	$0,1$ ms	$3,2$ ms	$24,3$ ms	$102,4$ ms	$312,5$ ms	$777,6$ ms
2^n	~ 1 µs	~ 1 ms	~ 1 s	~ 18 min20 s	~ 13 jours	~ 36 années et 6 mois

TABLE 1. Comparaison de diverses fonctions de complexité pour un ordinateur effectuant 1 milliard d'opérations par seconde.

Taille de l'instance la plus large que l'on peut résoudre en 1 heure

Fonction de complexité	Avec un ordinateur actuel	Avec un ordinateur 100 fois plus rapide	Avec un ordinateur 1000 fois plus rapide
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

TABLE 2. Comparaison de diverses fonctions de complexité.

Lorsqu'on ne trouve pas d'algorithme polynomial pour résoudre un pb : on peut trouver un algorithme exponentiel en examinant *toutes* les possibilités. Mais comment montrer qu'il n'existe pas d'algorithme efficace (polynomial)?
Pas de preuve directe (actuellement) qu'un problème est *intrinsèquement intractable*. Mais concept de *problème **NP**-complet*. Théorie de la **NP**-complétude fournit des techniques variées pour prouver qu'un problème est aussi difficile qu'une grande quantité d'autres problèmes, pour lesquels aucune méthode efficace n'a été trouvée jusque là.
Prouver qu'un problème est **NP**-complet est le début du travail algorithmique: ne plus se focaliser sur la recherche d'un algorithme exact et efficace et chercher des algorithmes efficaces résolvant divers cas particuliers (algorithmes sans garantie sur le temps d'exécution, mais en général rapides ; relaxer le problème et chercher un algorithme rapide qui résout; commencer par une solution et l'améliorer ...)

2.1 Quelques définitions

Problème de décision : existe-t-il une solution qui satisfait une certaine propriété ? – Résultat: Oui/Non
Un *problème de décision P est dans **NP*** si, lorsque la réponse est oui, il existe un certificat (c'est-à-dire une information supplémentaire) et un algorithme polynomial qui permet de vérifier que la réponse est oui , sans pour autant être capable de trouver cette réponse.
Le sigle **NP** ne signifie pas non polynomial mais non déterministiquement polynomial sur une machine de Turing.

Principale question ouverte : P=NP ? (prix d'1 millions de dollars offert par la Fondation Clay)

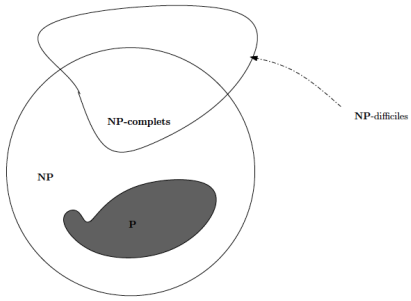
*Un problème de décision est dit **NP**-complet* si l'existence d'un algorithme polynomial le résolvant implique l'existence d'un algorithme polynomial pour tout problème **NP**. A ce jour, on ne connaît pas d'algorithme polynomial résolvant un problème **NP**-complet (mais on connaît beaucoup de problèmes **NP**-complets).

Exemples P et NP
– Le problème du *cycle eulérien* (passant une seule fois par toutes les arêtes) est dans **P** (vérifier que tous les noeuds de G sont d'arité paire).
– Le problème du *cycle hamiltonien* (passant une seule fois par tous les sommets): on ne sait pas si ce problème est dans **P** ou pas. En revanche, il est facile de voir que *ce problème est dans **NP*** car si la

réponse est positive, alors l'algorithme (polynomial) qui consiste à suivre le cycle hamiltonien permet de prouver que la réponse est bien positive. Il a été également démontré que *ce problème est **NP**-complet.*

Notion qui permet de caractériser des problèmes difficiles qui ne sont pas des problèmes de décision : un *problème est **NP**-difficile* si savoir le résoudre en temps polynomial impliquerait que l'on sait résoudre un problème (de décision) **NP**-complet en temps polynomial. Les problèmes **NP**-difficiles sont donc dans un sens plus dur que les problèmes **NP**-complets.

Par exemple le problème du voyageur de commerce (TSP) est **NP**-difficile.
Input $G =< S, A, w >$
Output : un cycle hamiltonien de coût minimum



2.2 Problème d'optimisation

Problème de décision vs d'optimisation :
– *Décision* : existe-t-il une solution qui satisfait une certaine propriété ? – *Résultat*: Oui ou Non
– *Optimisation* : parmi les solutions qui satisfont une certaine propriété, trouver celle qui optimise une certaine fonction de coût. – *Résultat*: une solution réalisable optimale
$\mathcal{S}$: l'ensemble des solutions possibles. $f : \mathcal{S} \to R$ fonction de coût. Trouver $S*$ tq $f(S*) < f(S), \forall S \in \mathcal{S}$.

Heuristique Lorsque la détermination d'une solution exacte prend trop de temps de calcul trouver un algorihme efficace pour déterminer une "bonne" solution (et non la meilleure).
Souvent *solution approchée*, la meilleure possible dans un temps raisonnable. Mais compromis qualité/temps souvent impossible à évaluer numériquement. Quand on sait évaluer *de manière théorique* la qualité de la solution, on parle d'*algorithme d'approximation.* Si évaluation *expérimentale*, on parle d'*heuristique.*

L'objectif d'une heuristique est de calculer efficacement des 'bonnes' solutions mais sans garantie d'optimalité!
Qu'est ce qu'une bonne heuristique ? (manque de résultats théoriques)
– de complexité raisonnable (idéalement polynomiale mais en tout cas efficace en pratique),
– simple à mettre en oeuvre,
– robuste : fournit le plus souvent une solution proche de l'optimum,
– probabilité faible d'obtenir une solution de mauvaise qualité.

3 Recherche locale

Local Search Heuristic : i) Commencer avec une solution initiale de l'espace de recherche ii) 'itérativement' visiter une solution 'voisine'. NB incomplète: pas de garantie de trouver une solution (optimale).

Algorithm 2 Recherche locale

```
S ← solution initiale ;
while Non Condition d'arrêt do
    N(S) ← générer des solutions voisines ;
    S' ← choisir une solution dans N(S) ;
    S ← S'
end while
```

- Générer une solution initiale S
  – Random (pas facile en général)
  – Heuristic (e.g., greedy)

  NB: Une solution initiale de meilleure qualité ne produit pas forcément meilleure solution à la fin !

- Voisinage d'une solution $N(S)$ = ensemble de solutions qui diffèrent peu de S : opérateur de voisinage $N(S)$ par rapport à une représentation du problème/solution :
  – graphe qui diffère d'1 arête
  – graphe ou l'on remplace un "triangle" par arete par rapport au centre
  – représentation binaire qui diffère d'1 bit
  – permutation qui diffère d'une inversion

- Choisir une solution dans $N(S)$ qui améliore la fonction de coût $f : \mathcal{S} \to \mathcal{R}$. La sol S' est choisie de façon locale à partir de la solution courante S:
  – aléatoirement – peu efficace
  – Choisir le meilleur voisin parcourir tous les voisins ! Coût en temps
  – Choisir le premier voisin S' tel que $f(S') < f(S)$ – souvent plus efficace

  L'étape la plus coûteuse est en général l'évaluation → Calculer la différence d'une solution voisine S' par rapport à S plutôt que recalculer le coût total de S'.

- Condition d'arrêt ?
  – nombre fini d'itérations
  – pas d'amélioration possible (Optimum local) (nombre fini d'itérations sans amélioration).
  On a alors solution S'' tel que $f(S'') < f(S')$ pour tout S' dans $N(S)$.

Trajectoire de recherche : $S_0 \to S_1 \to S_2 \to \ldots \to S_k$, S_k dépend de S_0, du voisinage N, de fonction f.
S'échapper des Optima locaux !
– recommencer avec une nouvelle solution initiale
– sur un optimum local, faire des sauts vers des solutions voisines ayant un coût moins bon.
Mais pas de garantie que cela soit effectif !

Bonne recherche locale : équilibre entre
– *Intensification*(itérativement, de façon gloutonne, augmenter la qualité de la solution (hill-climbing))
– *Diversification* (éviter de stagner dans une région (e.g. Optima locaux), random local search par ex.)

Dans une recherche locale, les optima locaux dépendent du choix de f et du voisinage N. Jouer sur N :
– Utiliser un plus grand voisinage : cas idéal = voisinage exact : tout optimal local est un optimal global. Mais trop grand pour être cherché efficacement (temps).
– Changer de voisinage: un optimum local pour un voisinage N n'est pas forcément un optimum local pour un voisinage N'