Algorithmique avancée

Denis TRYSTRAM

Introduction à la complexité et à la résolution de problèmes difficiles

ENSIMAG 2A - filière Alternants 2019

Constat et positionnement

La complexité est une notion intuitive dont chacun a fait un jour l'expérience à son niveau. Par exemple, lorsque l'on implémente un algorithme *brute force* pour résoudre un problème tel que le problème de la partition de n entiers ou tel que le voyageur de commerce, nous faisons face très rapidement à des problèmes de temps de résolution prohibitifs (même pour des instances avec peu de tâches ou peu de machines) ou une occupation mémoire considérable.

Il s'agit de problèmes *difficiles*, mais est-ce dans la nature même du problème ou parce que l'on n'a pas trouvé de bonne manière de faire ?

Les informaticiens, dont un des objectifs est de s'occuper de *calcul*, ont eu besoin de formaliser cette notion, pour classifier les problèmes qui peuvent être résolus grâce à un ordinateur, ou à défaut, pour en fournir de "bonnes" solutions approchées.

1 Modèles de calcul

Les modèles de calcul ont intéressé les logiciens et mathématiciens depuis le début du siècle dernier, avant même l'apparition des ordinateurs. Depuis David Hilbert en 1890, jusqu'à Alan Turing ou Kurt Gödel dans les années trente (du siècle dernier)... Aujourd'hui, il est clair, au moins intuitivement, que les machines calculent des *fonctions*, mais, qu'est-ce que cela veut dire exactement ?

Pour un mathématicien, une fonction f est un ensemble de couples (a, b) tels que si $(a, b) \in f$ avec $a \in A$ et $b \in B$ et $(a, c) \in f$ alors $b = c$. Habituellement, on note plutôt $f : A \to B$ et $f(a) = b$.

Un programmeur préfère plutôt définir les fonctions par des *algorithmes*, c'est-à-dire des méthodes/procédures qui les calculent. Ainsi, étant donné a on cherche une méthode qui calcule $b = f(a)$. Ces deux façons de considérer les fonctions (aspects que l'on pourrait qualifier respectivement de reconnaissance et calculatoire) sont en fait équivalentes...

La question importante est de déterminer quelles fonctions sont calculables par une machine donnée, et pour cela, il est nécessaire de définir ce

qu'est un *modèle de calcul* et si possible, un qui soit universel (c'est-à-dire, qui est valable quelle que soit la machine considérée)... Evidemment la réponse dépend du langage choisi disponible sur une machine (langage de programmation). Tous les langages de programmation "classiques" ont la même puissance de calcul au sens où ils permettent de calculer les mêmes fonctions.

Le modèle de référence des calculateurs universels est la *machine de Turing*, on considère plus pragmatiquement le modèle équivalent des RAM (random access machines).

2 Les ensembles $\mathcal{P}$ et $\mathcal{NP}$

La notion de complexité a été introduite pour qualifier la nature plus ou moins ardue des problèmes qui admettent une solution sous forme de *procédure effective* (i.e. calculable sur un ordinateur universel).

La classe de complexité qui nous intéresse *a priori*, $\mathcal{P}$, correspond aux problèmes pouvant être résolus en temps polynomial. Si tous les algorithmes polynomiaux ne sont pas efficaces (un algorithme de complexité n^{100} est inutilisable en pratique), un algorithme exponentiel, de complexité 2^n par exemple, est clairement inefficace (encore moins en n^m). Malheureusement une très large proportion de problèmes "intéressants" ont peu d'espoir d'admettre un algorithme de résolution polynomiale. La plupart de ces problèmes appartiennent à une classe de complexité plus vaste, $\mathcal{NP}$. La différence entre $\mathcal{P}$ et $\mathcal{NP}$ est que pour un problème de $\mathcal{P}$, il est possible de *trouver* en temps polynomial la réponse à **toutes** les instances, tandis que pour un problème de $\mathcal{NP}$ il est possible de *vérifier* en temps polynomial qu'une réponse est correcte.

La théorie de la complexité se définit sur les problèmes de décision (dont la réponse est binaire, "oui" ou "non"), et cherche à déterminer le plus petit temps d'exécution nécessaire à un algorithme pour décider un problème. La modélisation adoptée précédemment de la notion de résolution de problème et de l'exécution d'un algorithme conduit à reformuler formellement notre objet d'étude comme la détermination du plus petit temps de calcul nécessaire à une machine de Turing pour *décider si la réponse est "oui" ou "non"*. Rappelons que la résolution se fait par le choix d'un codage *naturel* de ses instances. La complexité d'un problème ne sera ainsi définie qu'à un polynome près, dépendant du choix du codage naturel.

Les complexités en temps et en espace (mémoire) sont ainsi définies par rapport au modèle de la machine de Turing comme le nombre de transitions et le nombre de cases mémoires utilisées. Nous exprimons la complexité en fonction de la taille de l'instance à traiter, c'est-à-dire en fonction de la taille du mot en entrée.

La classe $\mathcal{NP}$ est définie par rapport aux machines de Turing non déterministes (NTM). Rappelons qu'une NTM accepte un mot simplement si il existe une exécution acceptant ce mot. Ainsi, une NTM résout le problème en temps $f(n)$ si :

- Pour tout mot w d'entrée, il existe une exécution acceptant w en au plus $f(|w|)$ transitions.
- Pour tout mot w qui n'appartient pas aux entrées, aucune exécution (quelle que soit sa longueur) ne conduit à un état accepteur de la machine.

Attention, soulignons que la complexité d'un problème est défini par rapport à un codage naturel de ses instances. Considérons par exemple le problème du test de primalité d'un nombre :

PRIME
Instance : un entier N
question N est-il premier ?

Il existe un algorithme connu pour répondre à cette question en temps $O(\sqrt{N})$ (le crible d'Erathostène). Mais un codage naturel de PRIME consiste à représenter N en binaire. Le mot d'entrée étant ainsi codé sur $\log_2(N)$ bits, la complexité de l'algorithme est en fait exponentielle ! Ce problème est un exemple de problème dont il n'est pas facile simplement de prouver qu'il est dans $\mathcal{NP}$ (alors que son complémentaire – reconnaitre s'il n'est pas premier – est très simple en multipliant ses diviseurs...).

Une autre caractérisation possible $\mathcal{NP}$ consiste, plutôt qu'exhiber un algorithme polynomial non déterministe reconnaissant un problème, à exhiber un certificat de positivité *concis* (polynomial). C'est-à-dire que pour tout mot d'entrée, il existe un preuve d'appartenance vérifiable en temps polynomial (en d'autres termes, on peut vérifier qu'une solution donnée est bien une solution).

Définition $\mathcal{NP}$ est l'ensemble des problèmes tels que pour tout mot w d'entrée, il existe une preuve d'appartenance π_w de w vérifiable en temps (déterministe) polynomial en $|w|$.

Ainsi, pour le problème HAMILTONIANCIRCUIT (HC) décider l'existence d'un cycle Hamiltonien dans un graphe est simple :

HC
Instance: Un graphe $G = (V, E)$
question Existe-t-il un cycle Hamiltonien ? (c'est-à-dire un cycle passant une fois et une seule par chaque sommet).

Le problème HC appartient à $\mathcal{NP}$. Considérons comme codage naturel la matrice d'adjacence du graphe. Un certificat de positivité consiste à donner une permutation des sommets. Ce certificat est de taille $\mathcal{O}(n \log n)$ qui correspond au codage des nombres successifs de cette permutation. On peut vérifier en temps $\mathcal{O}(n^2)$ que la permutation correspond à un cycle du graphe. Ce certificat est bien polynomial en la taille de l'instance $\Theta(n^2)$.

3 Réductions

Le principe de l'étude de la complexité est de classifier les problèmes par rapport au critère de temps d'exécution sur une machine de Turing, et comme les choses sont définies à un polynome près, sur toute machine raisonnable comme les RAMs. Les deux classes $\mathcal{P}$ et $\mathcal{NP}$ que nous avons définies ne semblent pas assez fines pour discriminer la difficulté des problèmes. Nous aimerions introduire une relation d'ordre sur les problèmes, signifiant qu'un problème est plus facile qu'un autre. Cette relation d'ordre est définie par la *réduction polynômiale*.

La réduction τ transforme toutes les instances positives Π en instances positives de Π', et toutes instances négatives de Π en instances négatives de Π'. L'existence d'une réduction polynômiale de Π vers Π' montre que Π' est au moins aussi difficile que Π. En effet, si Π peut être résolu en temps polynomial, alors Π' peut l'être aussi; si par contre Π requiert un temps exponentiel, alors Π' ne peut être résolu par un algorithme polynomial. Notons bien que le sens premier de la réduction de Π vers Π' est encore plus fort : une réduction prouve qu'à une transformation polynomiale près des instances, c'est-à-dire à un codage naturel près de Π, le problème Π est simplement un sous-problème de Π'...

D'un point de vue algorithmique, une réduction est un *preprocessing* des instances du problème Π, qui permet d'utiliser tout algorithme polynomial de résolution pour Π' pour résoudre Π en temps polynomial.

On notera $\Pi \propto \Pi'$ si il existe une réduction polynomiale de Π vers Π'. Nous dirons que Π se réduit à Π'.

3.1 Deux exemples de réductions

Détaillons un exemple de réduction polynomiale du problème de décision qui cherche à déterminer si un graphe possède une chaîne hamiltonnienne à partir du problème du cycle hamiltonnien.

Cycle et chemin hamiltonnien

HP (HamiltonianPath)
Instance. un graphe $G = (V, E)$

Question. Est-ce que le graphe possède une chaîne hamiltonnienne dans G (chaîne qui passe une fois et une seule par tous les sommets du graphe) ?

Pour montrer que nous avons la réduction HP $\propto$ HC, considérons une instance de HP, c'est-à-dire un graphe G.

A partir de G nous construisons une instance particulière $\tau(G)$ de HC en ajoutant à G un nouveau sommet x relié à tous les autres : $\tau(G) = (V \cup \{x\}, E \cup (x, y) \forall y \in V)$.

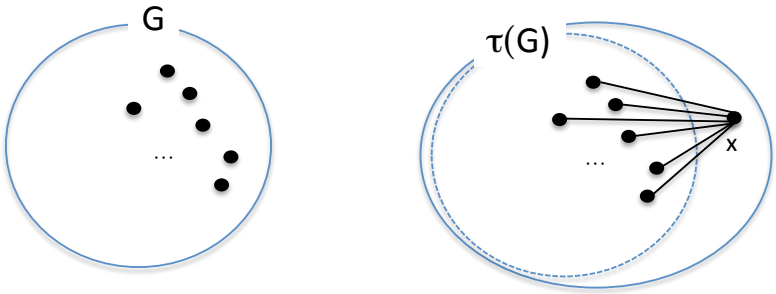


Figure 1: Réduction de HP vers HC.

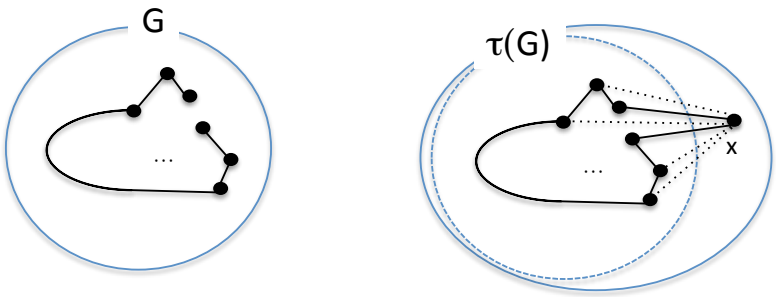


Figure 2: Un chemin hamiltonnien de G donne un cycle de $\tau(G)$.

La transformation τ est évidemment polynomiale. Montrons que c'est une réduction, c'est-à-dire que G admet une chaîne hamiltonnienne si et seulement si $\tau(G)$ possède un cycle hamiltonnien.

Si G possède une chaîne hamiltonnienne φ, alors le cycle $x\varphi x$ est hamiltonnien dans $\tau(G)$. Réciproquement, si $\tau(G)$ admet un cycle hamiltonien, son sous-graphe privé de x, G, possède une chaîne hamiltonnienne.

On peut également établir que nous avons la réduction du cycle hamiltonnien au circuit hamiltonnien, $HC \propto HP$. Ce résultat n'est pas aussi immédiat, même s'il semble facile de déduire une chaîne hamiltonnienne à

partir d'un cycle hamiltonnien, cela ne suffit pas à définir une réduction. Considérons la transformation τ suivante d'une instance G de HC vers une instance $\tau(G)$ de HP :

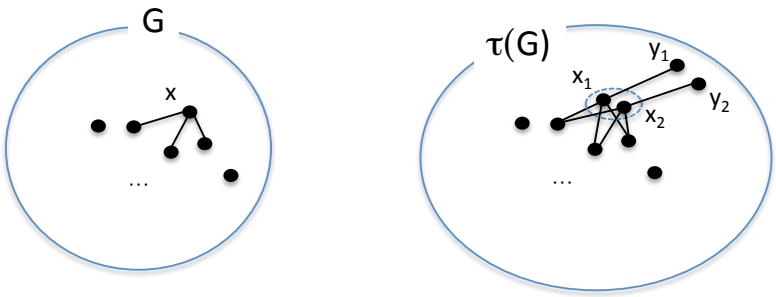


Figure 3: Réduction de HC vers HP.

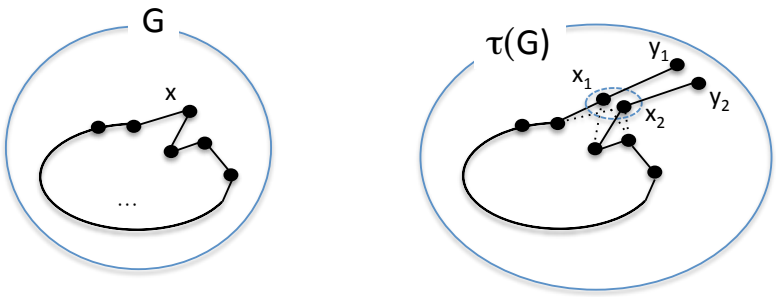


Figure 4: Un cycle hamiltonnien de G donne un chemin de $\tau(G)$.

On duplique un sommet quelconque x du graphe G en (x_1, x_2), puis on relie x_1 et x_2 respectivement à deux nouveaux sommets y_1 et y_2 comme c'est indiqué dans la figure 3. Cette transformation est bien polynomiale.

C'est une réduction : G admet un cycle hamiltonnien si et seulement si $\tau(G)$ possède une chaîne hamiltonnienne.

En effet, si G possède un cycle hamiltonnien, la chaîne formée de l'arrête y_1 à x_1, de la partie du cycle jusqu'à un voisin de x_1, puis des arrêtes de ce voisin à x_2 et celle de x_2 à y_2 est une chaîne hamiltonnienne de $\tau(G)$.

Réciproquement, s'il existe une chaîne hamiltonnienne φ dans $\tau(G)$, elle est nécessairement de la forme $y_1 x_1 \psi x_2 y_2$. Alors $x\psi x$ est un cycle hamiltonnien sur G.

Nous venons de montrer que les problèmes HC et HP sont équivalents : nous pouvons transformer tout algorithme polynomial pour HC en un algorithme polynomial pour HP, et inversement.

Voyageur de commerce

Considérons deux problèmes voisins, les versions de décision du problème du voyageur de commerce (D-TSP) et de l'existence d'un circuit hamiltonien dans un graphe.

D-TSP
Instance. un ensemble V de villes, la matrice des distances inter-villes $(d_{i,j})$ et une constante k.
Question. Déterminer s'il existe un parcours fermé, passant une fois et une seule par toutes les villes, de longueur inférieure à k.

Il est facile de démontrer que ce problème admet un algorithme de résolution exponentiel en considérant toutes les permutations possibles et en comparant la somme de la longueur de tous les tours. A ce jour, il n'existe pas d'algorithmes polynomiaux connus pour résoudre ce type de problèmes, et nombreux sont les informaticiens qui pensent qu'il n'en existe sans doute pas... (C'est là une des célèbres questions ouvertes de l'Informatique à un million de dollars, avis aux amateurs !).

On peut réduire D-TSP à partir de HC (HC $\propto$ D-TSP).

Décrivons un algorithme polynomial qui transforme une instance quelconque de HC en une instance positive de D-TSP si et seulement si l'instance de HC est positive.

La réduction est la suivante : l'ensemble des villes correspond aux sommets du graphe G, les distances sont données par $d_{i,j} = 1$ si i et j sont reliés, 2 sinon. La constante k est égale à n (nombre de villes).

Cette transformation est polynomiale de manière évidente.

Si G possède un cycle hamiltonnien alors ce cycle correspond à une tournée (qui passe par toutes villes une fois et une seule) et sa longueur est n, donc $\tau(G)$ est positive.

Réciproquement, une instance positive de D-TSP dans $\tau(G)$ est transformée en une instance positive de HC. En effet, la solution de D-TSP possède par définition des arêtes de coût unitaire puisque la longueur totale est n, ce qui correspond bien à un cycle hamiltonnien de G.

4 Problèmes NP-complet

Nous venons de voir qu'il existe des problèmes dans $\mathcal{P}$ et $\mathcal{NP}$, dans un soucis de classification, muni de l'outil des réductions polynomiale, il est naturel de s'intéresser à la question de la structuration de $\mathcal{NP}$. En particulier, il existe un plus grand élément au sens de ces réductions, c'est-à-dire une classe qui contient les problèmes plus difficiles que tous les autres (c'est la classe NP-complet). Cook a exhiber un tel problème en 1971 avec SAT (satisfability)

en montrant que la résolution de tout problème pouvait se coder sous forme d'expression logique en forme normale conjonctive. Résoudre le problème revient ainsi à SAT. La preuve est technique, elle est omise ici.

4.1 Présentation du problème SAT

Le problème SAT est un problème de logique propositionnelle (encore appelée logique d'ordre 0 ou logique des prédicats). Rappelons qu'un prédicat est défini sur un ensemble de variables logiques, à l'aide des 3 opérations élémentaires que sont la négation NON ($\neg x$ que nous noterons aussi $\bar{x}$), la conjonction ET ($x \wedge y$) et la disjonction OU ($x \vee y$). Un littéral est un prédicat formé d'une seule variable (x) ou de sa négation $\bar{x}$.

Une clause est un prédicat particulier, formée uniquement de la disjonction de littéraux, par exemple $C = x \vee \bar{y} \vee z$. Une formule est sous forme normale conjonctive si elle s'écrit comme la conjonction de clauses. Le problème SAT consiste à décider si une formule en forme normale conjonctive est satisfiable, c'est-à-dire si il existe une assignation τ aux variables telles que toutes les clauses sont *true*.

SAT (Satisfiability)
Instance. m clauses C_i formées à l'aide de n variables logiques (littéraux)
Question. la formule $\Phi = C_1 \wedge C_2 \wedge ... \wedge C_m$ est-elle satisfiable ?[1]

Par exemple, la formule $(x \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{y}) \wedge (x \vee \bar{z})$ est satisfaite avec l'assignation $\gamma(x) = \mathit{true}$ et $\gamma(y) = \mathit{false}$. Il est facile de se convaincre que le problème SAT est dans $\mathcal{NP}$. Un certificat de positivité consiste à donner une assignation des variables, codable sur un vecteur de n bits. La vérification que toutes les clauses sont satisfaites est alors clairement polynomiale (plus précisément, dans $\Theta(nm)$).

Comme nous l'avons évoqué plus tôt dans ce chapitre, Cook a montré en 1971 que ce problème est NP-complet en codant l'exécution de tout algorithme sur une machine de Turing non-déterministe par une expression en forme normale conjonctive.

5 Une première série de variante : kSAT

On note kSAT la variante du problème SAT où toutes les clauses ont exactement k littéraux ($k \geq 2$). La variante obtenue pour $k = 3$ est particulièrement utile, on peut établir le résultat simple suivant.

Proposition. 3SAT est NP-complet.

[1] ou encore : existe-t-il une fonction d'interprétation qui rende la formule vraie ?

preuve. Remarquons tout d'abord qu'il est facile de vérifier que 3SAT $\in \mathcal{NP}$ (c'est un cas particulier de SAT qui est dans $\mathcal{NP}$).

L'idée de la réduction à partir de SAT est simplement de réécrire chaque clause comme une clause de cardinalité 3.

- Pour les clauses de deux littéraux, on rajoute une variable et on introduit deux nouvelles clauses qui ne changent pas l'évaluation de la formule.

  Par exemple pour le cas d'une clause de cardinalité 2, $(x \vee y)$, on rajoute la variable z, et on réécrit la clause comme $(x \vee y \vee z) \wedge (x \vee y \vee \bar{z})$.

  Pour les clauses formées d'une seule variable, on rajoutera de même deux variables...

- Pour les clauses de cardinalité p strictement supérieure à 3, on procède avec la même idée de rajouter des variables qui conservent la valeur logique à l'expression. Ainsi, on rajoute $p-3$ variables et autant de clauses.

  Par exemple pour une clause formée de 5 littéraux $(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5)$, on introduit 2 nouvelles variables y_1 et y_2 :

  $(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) = (x_1 \vee x_2 \vee y_1) \wedge (\bar{y_1} \vee x_3 \vee y_2) \wedge (\bar{y_2} \vee x_4 \vee x_5)$

  La réduction proprement dite est alors simple à vérifier, elle est laissée en exercice.

5.1 Prouver la NP-complétude d'un problème

Comment prouve-t-on qu'un problème est dans NP-complet (on dit plus simplement par abus de langage qu'il est NP-complet) ?

- Montrer qu'il est dans $\mathcal{NP}$
- Choisir un problème dans NP-complet (il en existe aujourd'hui un très grand nombre) et construire une réduction polynomiale adéquate

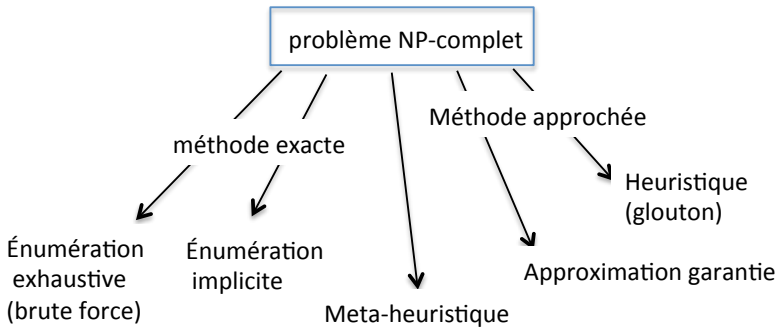
Les problèmes NP-complet que nous allons utilisés dans ce cours sont les suivants :

- 2-Partition
- Knapsack
- SubSetSum
- Ordonnancement sur 2 machines
- 3SAT
- HC ou HP
- TSP

6 Résoudre des problèmes difficiles

Il y a une conséquence pratique importante des résultats des sections précédentes : il est raisonnable de diriger nos efforts vers d'autres approches que la résolution (exacte) exhaustive des problèmes NP-complets.

Résolution exacte (avec énumération implicite), méthode approchée, avec garantie ou non. Quelle que soit la méthode choisie, il s'agit toujours d'un compromis entre rapidité et "qualité" des solutions...



Les meta-heuristiques (tabu search, recuit simulé, algorithmes génétiques, etc.) sont des méthodes généralistes, lourdes à mettre en œuvre, qui nécessitent de nombreux réglages. Elles sont efficaces au sens où elles peuvent fournir aux spécialistes des solutions de bonnes qualités.

7 Conclusion

Nous voilà prêt à résoudre toute sorte de problèmes ! L'analyse de complexité donne souvent de bonnes intuitions pour la résolution et surtout, elle permet d'identifier les limites de ce que l'on peut attendre d'un algorithme de résolution.