Problème des mariages stables

François Schwarzentruber

Modalités

- Biblio : [Papadimitriou et al.], [Cormen et al.], [Kleinberg-Tardös]
- Cours : mardi après-midi de 13h15 à 14h45 à l'ENS Rennes (amphi) TD : jeudi matin de 8h à 10h à Beaulieu avec Theo Loosekoot et Kilian Barrere
- Terminal : devoir sur table 1h30
- Contrôle continu : pour chaque séance de TD, vous rendez la correction d'un exercice (à rendre une semaine après)

1 Motivation

But : marier les $a \in A$ et les $b \in B$.

A	B
hommes	femmes
étudiant.e.s	encadrant.e.s de stage
candidats.es	postes
donneurs.ses de reins	receveurs.ses de reins

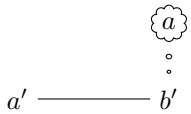
Chaque $a \in A$ exprime ses préférences sur les éléments de B.
Chaque $b \in B$ exprime ses préférences sur les éléments de A.

2 Modélisation

Exemple 1 On considère l'ensemble $A = \{a_1, a_2, a_3\}$, l'ensemble $B = \{b_1, b_2, b_3\}$ et les préférences suivantes :

$$\begin{array}{lll} a_1 & : b_2 >_{a_1} b_1 >_{a_1} b_3 \\ a_2 & : b_1 >_{a_2} b_3 >_{a_2} b_2 \\ a_3 & : b_1 >_{a_3} b_2 >_{a_3} b_3 \end{array} \qquad \begin{array}{lll} b_1 & : a_1 >_{b_1} a_3 >_{b_1} a_2 \\ b_2 & : a_3 >_{b_2} a_1 >_{b_2} a_2 \\ b_3 & : a_3 >_{b_3} a_2 >_{b_3} a_1 \end{array}$$

On ne veut pas de "intérêt commun à changer" / instabilité.



Exemple 2 Voici un couplage avec une instabilité :

$a_1 b_1$ $a_2 b_2$ $a_3 b_3$

Le risque est que $a_1$et b_2 communiquent, divorcent et se marient ensemble :

a_1b_2 a_2b_1 a_3b_3

Ce dernier couplage est également instable.

Exemple 3 Voici des exemples de couplages stables :

a_1b_2 a_2b_3 a_3b_1 a_1b_1 a_2b_3 a_3b_2

3 Formalisation

Soit A et B deux ensembles finis à n éléments.

Définition 4 (couplage) Un sous-ensemble $S \subseteq A \times B$ est un *couplage* si tout élément de $A \cup B$ n'apparaît au plus qu'une seule fois dans un seul couple de S. (pas de polygamie)

Définition 5 (couplage parfait) Un couplage $S \subseteq A \times B$ est *parfait* si tout élément de $A \cup B$ apparaît exactement une fois dans un seul couple de S. (pas de polygamie, pas de célibataire)

On modélise les préférences des individus par des relations d'ordre totales strictes.

Définition 6 (collection de préférences) Une (A, B)-collection de préférences, notée $>$, est un couple de familles $((>_a)_{a\in A}, (>_b)_{b\in B})$ tel que :

- pour tout $a \in A$, $>_a$ est une relation d'ordre totale strict sur B ;
- pour tout $b \in B$, $>_b$ est une relation d'ordre totale strict sur A.

Définition 7 (instabilité) Étant donné $>$ et un couplage S, un couple $(a, b') \in A \times B$ est une $>$-instabilité pour S s'il existe $(a, b), (a', b') \in S$ avec $b' >_a b$ et $a >_{b'} a'$.

Définition 8 (couplage stable) Étant donné $>$, un couplage S est $>$-stable s'il est parfait et s'il n'y a pas de $>$-instabilité pour S.

Définition 9 (problème des mariages stables) Le problème des mariages stables est :

- entrée : deux ensembles finis A, B de même cardinal n, une (A, B)-collection de préférences $>$;
- sortie : un couplage $>$-stable $S \subseteq A \times B$ s'il en existe un.

4 Algorithme de Gale-Shapley

```
fonction GS(A, B, >)
    initialiser tous les éléments de A et de B comme libres
    tant que il existe a ∈ A libre, qui n'a pas fait de proposition à tous les b ∈ B
        choisir un tel a ∈ A
        soit b l'élément préféré de a, à qui a n'a pas encore fait de proposition
        //a fait une proposition à b
        si b est libre alors
            a et b s'engage
        sinon b est engagé avec un certain a′
            si a >_b a′ alors
                a et b s'engage
                a′ devient libre
    renvoyer l'ensemble S des couples engagés
```

Durant l'exécution de l'algorithme :

- les partenaires d'un a sont de pire en pire ;
- les partenaires d'un b sont de mieux en mieux.

Définition 10 On dit qu'un élément $a \in A$ se fait rejeter quand : soit il se propose à un b mais b a déjà mieux. Soit, il se fait larguer par b.

5 Exemple d'exécution

Avec magnets.

6 Terminaison

Théorème 11 $\mathsf{GS}(A, B, >)$ termine après au plus n^2 itérations de la boucle **tant que** où $n = \#A = \#B$.

DÉMONSTRATION. Exhiber un variant, c'est-à-dire un entier positif qui décroit strictement à chaque itération de boucle.

Plus généralement, un élément qui décroit strictement où la relation $<$ est bien fondée. On rappelle qu'une relation $<$ bien fondée s'il n'y a pas de suites infinies $(x_n)_{n\in\mathbb{N}}$ avec $x_{n+1} < x_n$ pour tout n.

(le choix du variant est subtil : par exemple le nombre d'individus libres n'est pas un variant, car il peut rester inchangé d'une itération à l'autre)

Soit $C(t)$ l'ensemble des couples (a, b) où a n'a pas encore fait de proposition à b entre le début de l'algorithme et la fin de la $t^{\text{ème}}$ itération de la boucle **tant que** .

$$\#C(0) = n^2 \text{ et } \#C(t+1) < \#C(t)$$

Donc il ne peut y avoir plus de n^2 itérations. ■

7 Correction

Remarque 12 L'algorithme est sous-spécifié. Sur une même entrée $(A, B, >)$, il admet plusieurs exécutions, que l'on peut résumer sous la forme d'un arbre, appelé arbre de calcul.

noeud : configuration

branche : une exécution

En fait, les algorithmes sont quasiment toujours sous-spécifiés (asynchronisme dans une architecture multi-coeurs, délai de réponse d'un serveur, etc.)

Théorème 13 Toute exécution de $\mathsf{GS}(A, B, >)$ renvoie un couplage S qui est $<$-stable.

DÉMONSTRATION. Exhiber un ou plusieurs invariants, i.e. des propriétés qui restent vraies tout au long de l'algorithme (ou d'une partie)

1. Montrons que S est un couplage.

   (I1): "l'ensemble des couples engagés est un couplage".

   (I1) est un invariant car vrai au début car $\emptyset$ est un couplage; conservé par une itération.

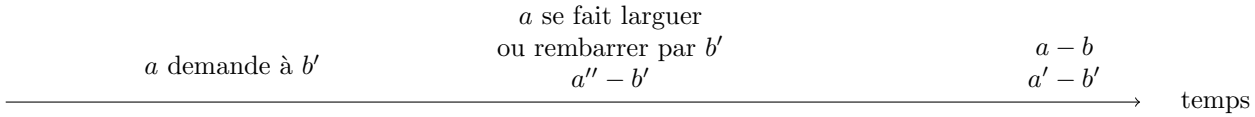
2. Montrons que le couplage S renvoyé est parfait. Par l'absurde, supposons que S n'est pas parfait. Il existe alors $a \in A$ libre. Mais comme on a quitté la boucle tant que, a a fait des propositions à tous les b. Mais alors, tous les b sont engagés (dès qu'un b a reçu une proposition, il est engagé avec quelqu'un pour toujours !). Donc le couplage est parfait. Contradiction.

   Du coup, la condition de la boucle **tant que** devient "**tant que** il existe un a libre".

3. Montrons que S est stable. Par l'absurde, supposons qu'il existe une instabilité, i.e. qu'il y a $(a, b), (a', b') \in S$, $b' >_a b$ avec $a >_{b'} a'$.

   La dernière proposition réussie de a était faite à b. Comme $b' >_a b$, l'agent a eut fait une proposition à b' (si a fait une proposition à un certain b, il a fait une proposition à toutes les personnes avant ce b dans sa liste de préférence).

   Comme a n'est pas avec b', b' a ensuite rejeté a pour un certain a'' que b' préfère, $a'' >_{b'} a$. Soit b' était avec a'' lors que a a fait sa proposition, soit a s'est fait largué plus tard... cela n'a pas trop d'importance.



Comme les partenaires de b' sont de mieux en mieux, et que a' est le partenaire final de b', on a $a' \geq_{b'} a''$, i.e. soit $a'' = a'$, ou alors $a' >_{b'} a''$. Par transitivité de $>_{b'}$, on a $a' >_{b'} a$. Contradiction . S est bien stable.

■

Corollaire 14 Étant donné A, B et $>$, il existe toujours un couplage $>$-stable.

8 Implémentation

Théorème 15 On peut implémenter GS en temps $O(n^2)$, le corps de la boucle s'exécutant en $O(1)$.

Démonstration. Voici comment procéder.

- On suppose $A = B = \{1, \ldots, n\}$;
- Un $n \times n$-tableau $aPref$ donne les préférences des $a \in A$. Plus précisément, $aPref[a, i]$ est le $i^{\text{ème}}$ élément de B préférée par a.
- Pareil pour $bPref$.
- Liste chaînée pour stocker les $a \in A$ libres;
- Un tableau $next$ pour savoir à qui demande ensuite. $next[a]$ est le rang du $b \in B$ suivant à qui a va faire une proposition.
  - Au début, $next[a] = 1$ pour tout $a \in A$.
  - L'agent a propose à $aPref[a, next[a]]$, puis on incrémente $next[a]$ à chaque fois qu'il propose.
- Si b est libre, $current[b] = free$, sinon $current[b]$ est le partenaire courant de b.
- Malheureusement, le test $a >_b a'$ est en $O(n)$ avec $bPref$. Pour avoir ce test en $O(1)$, on précalcule une table $ranking$ où $ranking[b, a]$ est le rang de a dans les préférences de b, i.e. le i tel que $bPref[b, i] = a$.

■

9 Pseudo-code détaillé

```
GS(A, B, <)
        init()
        declarerLibreA(A)
        declarerLibreB(B)

        Tant que existeLibreA()
                choisir a libre
                Soit b := getElementPrefereeNonPropose(a)

                Si estLibreB?(b)
                        engage(a,b)
                Sinon
                        a' := partenaireA(b)
                        Si a' <b a
                                engage(a,b)
```

```
declarerLibreA(A)
        for(a in A)
                L := a::L

declarerLibreB(B)
        for(b in B)
                current[b] = .


existeLibreA()
        retourner L != null



init()
        for(a in A)
                Next[a] := 1
        for(b in B, i in {1, ..n})
                ranking[b, BPref[b, i] = i


getElementPrefereeNonPropose(a)
        APref[a, Next[a]]
        incrémenter Next[a]
```

10 Algorithme injuste

Exemple 16

$$\begin{array}{ll} a : b >_a b' & b : a' >_b a \\ a' : b' >_{a'} b & b' : a >_{b'} a' \end{array}$$

L'algorithme GS donne :

$$\begin{array}{cccc} a & b & a & - \; b \\ a' & b' \Rightarrow & a' & \quad b' \Rightarrow \end{array} \quad \begin{array}{c} a \; - \; b \\ a' \; - \; b' \end{array}$$

(et pareil si on commence par a')

Ainsi, le couplage stable $\begin{array}{c} a \; - \; b' \\ a' \; - \; b \end{array}$ n'est jamais calculé, mais le serait si c'était les $b \in B$ qui feraient les propositions.

En fait, toutes les exécutions calculent le même couplage : celui où chaque $a \in A$ est avec "son meilleur partenaire"... à définir formellement.

10.1 Algorithme favorisant les A

Définition 17 (partenaire valide) L'agent b est un partenaire valide de a s'il existe un couplage stable contenant (a, b).

Définition 18 (meilleur partenaire) Le meilleur partenaire de a est $best(a) = sup_{\leq_a}\{\text{partenaires valides de } a\}$
i.e. l'unique partenaire valide de a tel que pour tout $b' \in B$, b' partenaire valide de a implique $b' \leq_a best(a)$,

Théorème 19 Toute exécution de $\mathsf{GS}(A, B, >)$ retourne $S^* = \{(a, best(a)) \mid a \in A\}$.

Démonstration. Par l'absurde, supposons qu'il existe une exécution de $\mathsf{GS}(A, B, >)$ qui retourne $S \neq S^*$. Autrement dit, il existe $(a, b) \in S$ avec $b \neq best(a)$.

Fait 20 Il existe un $a \in A$ rejeté par un partenaire valide.

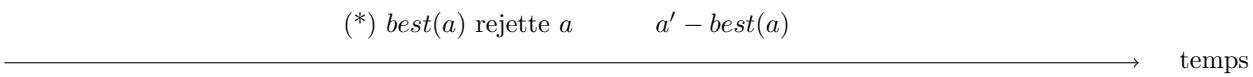
Démonstration. Prenons un a avec $(a, b) \in S$ avec $b \neq best(a)$. L'agent a est marié à b à la fin. L'élément b est donc valide pour a. Aussi, il s'est donc proposé en mariage à tous les éléments de B que a préfère à b, en particulier à $best(a)$, qui est le meilleur partenaire valide. Il s'est donc fait rejeté par $best(a)$. ■

Considérons le **premier moment** (*) de cette exécution où un certain $a \in A$ est rejeté par un partenaire **valide** pour lui (le fait **??** montre que de tel moment existe et donc on peut parler du premier moment où ça arrive). Ce partenaire valide est forcément $best(a)$: en effet, a a proposé dans l'ordre décroissant de ses préférences, et comme $best(a)$ est son partenaire valide préféré, ses rejets passés éventuels ne sont pas avec des partenaires valides.

Il y a deux situations de rejet de a au premier moment (*) :

- Soit a se propose à $best(a)$, mais $best(a)$ a déjà mieux : un beau a'.
- ou alors a se fait remplacer, car $best(a)$ a eu une meilleure offre par un beau a'.

Dans les deux cas, juste après le rejet de a, il y a un engagement $(a', best(a))$ avec $a' >_{best(a)} a$.

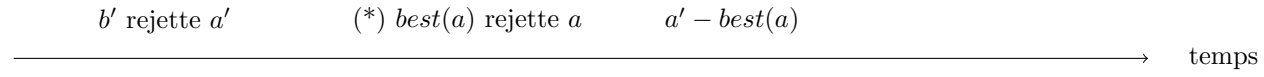


Par ailleurs, comme $best(a)$ est valide pour a, il existe un couplage stable S' contenant $(a, best(a))$. Soit b' le partenaire de a' dans S', i.e. tel que $(a', b') \in S'$. Le fait suivant conclut la démonstration avec une contradiction car S' est censé être stable.

Fait 21 On a $best(a) >_{a'} b'$. Et donc on a l'instabilité suivante dans S' :

Démonstration. Par l'absurde, supposons que $b' >_{a'} best(a)$.

1. En (*), on a $a' - best(a)$. Cela implique que a' a été rejeté par b' avant le moment (*), car a' propose par ordre de préférence décroissant, et donc a' s'est proposé à b'.
2. L'agent a' n'a pas été rejeté par un partenaire valide avant le moment (*) par définition de (*). Autrement dit, l'agent b' n'est pas valide pour a'.



3. Mais b' est valide pour a' car $(a', b') \in S'$.

Contradiction. ■

■

10.2 Algorithme défavorisant les B

On vient de voir que l'algorithme favorise les A, mais de façon symétrique, on peut montrer qu'il défavorise les B.

Définition 22 (partenaire valide) L'agent a est valide pour b s'il existe un couplage stable contenant (a, b).

Définition 23 (pire partenaire) Le pire partenaire valide pour b est $worst(b) = inf_{\leq_b}\{\text{partenaires valides de } b\}$.

Théorème 24 On a $S^* = \{(worst(b), b) \mid b \in B\}$.

Démonstration. Par l'absurde, supposons qu'il existe $a \in A$ tel que, en notant $b = best(a)$, on ait $a \neq worst(b)$. Alors il existe un couplage stable S' avec $(worst(b), b) \in S'$. Il existe $b' \neq b$ avec $(a, b') \in S'$. Dans S', on a l'instabilité :

■

11 Ensemble de tous les couplages stables

Fait 25 Il y a des exemples avec un nombre exponentiel de couplages stables en n.

Exemple 26 On prend $A = \{1, \ldots, n\}$ et $B = \{1, \ldots, n\}$ avec n pair.

Préférences des éléments de A :

$1 : 1\ 2\ ...$
$2 : 2\ 1\ ...$
$3 : 3\ 4\ ...$
$4 : 4\ 3\ ...$
$\vdots$
$n-1 : n\ n-1\ ...$
$n : n-1\ n\ ...$

Préférences des éléments de B :

$1 : ...\ 1$
$2 : ...\ 2$
$3 : ...\ 3$
$4 : ...\ 4$
$\vdots$
$n-1 : ...\ n-1$
$n : ...\ n$

Groupons les éléments de A par paire : 12, 34, ... Supposons que chaque élément d'une paire se marie soit avec leur soit avec leur premier choix, ou alors avec leur second choix.

Quand deux éléments de A d'une paire sont mariés à leur second choix, ils ne peuvent pas avoir leur premier choix car c'est le plus détesté des éléments de B.

Donc, tous ces mariages sont stables ! Il y en $2^{n/2}$.

12 Notes bibliographiques

Nous utilisons l'idée de Kleinberg et Tardos [KT06] : introduire l'algorithmique avec le problème des mariages stables et l'algorithme de Gale-Shapley. Gale et Shapley ont présenté l'algorithme dans [GS62].

Pour aller plus loin, vous pouvez consulter des ouvrages de référence dans le domaine du choix social comme [KMR16] ou [Man13], ainsi qu'un document en français écrit par Donald Knuth [Knu76].

References

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[KMR16] Bettina Klaus, David F. Manlove, and Francesca Rossi. Matching under preferences. In Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D. Procaccia, editors, *Handbook of Computational Social Choice*, pages 333–355. Cambridge University Press, 2016.

[Knu76] Donald Ervin Knuth. *Mariages stables et leurs relations avec d'autres problemes combinatoires: introduction a l'analysis mathematique des algorithmes-*. Les Presses de l'Université de Montréal, 1976.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[Man13] David F. Manlove. *Algorithmics of Matching Under Preferences*, volume 2 of *Series on Theoretical Computer Science*. WorldScientific, 2013.

Fast Fourier Transform

François Schwarzentruber

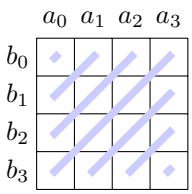
1 Motivation

1.1 Convolution discrète

La convolution est une opération qui intervient notamment en traitement du signal (filtre passe-bas, filtre passe-haut, reverb, etc.) et en traitement d'images (filtre gaussien, filtre de netteté, détection de contours, etc.). On donne ici la définition de la convolution discrète finie.

Définition 1 Soit $a = (a_0, \dots, a_{n-1})$ et $b = (b_0, b_1, \dots, b_{n-1})$ deux suites finies. La convolution discrète de a et b est la suite $c = (c_0, c_1, \dots, c_{2n-2})$ donnée par

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} = \sum_{i,j|i+j=k} a_i b_j$$



En fait, il s'agit de la même opération que le produit de deux polynômes A et B dont les coefficients sont donnés respectivement par a et b.

Proposition 2 (convolution dans produit de polynômes) Le produit des polynômes $A = \sum_{k=0}^{n-1} a_k X^k$ et $B = \sum_{k=0}^{n-1} b_k X^k$ est $AB = \sum_{k=0}^{2n-2} c_k X_k$ où $(c_0, c_1, \dots, c_{2n-2})$ est la convolution de $(a_0, \dots, a_{n-1})$ et $(b_0, b_1, \dots, b_{n-1})$.

Proposition 3 (utilisation de la convolution en probabilité) Soit X et Y deux variables aléatoires *indépendantes* à valeur dans $\{0, \dots, n-1\}$.

$\mathbb{P}(X + Y = k) = \sum_{i,j|i+j=k} \mathbb{P}(X = i)\mathbb{P}(Y = j)$

Définition 4 (problème algorithmique du calcul des coefficients du produit de deux polynômes)

- entrée : deux polynômes A et B décrits par leurs coefficients ;
- sortie : les coefficients de AB.

Proposition 5 Le calcul des coefficients de AB à partir des coefficients de A et B est en temps $\Theta(n^2)$.

2 Polynôme comme un type abstrait

Un polynôme peut être vu comme un type abstrait, avec des données, et des opérations : évaluation, addition, multiplication.

Il y a 3 façons de représenter un polynôme (implémentation) :

1. par une liste de n coefficients ;
2. avec la liste des $n-1$ racines $r_1, \dots, r_{n-1}$ et un facteur α ;
3. par les valeurs prises par le polynôme en n points deux à deux distincts.

On remarque pour les trois représentations, il y a exactement n nombres.

	représentation par n coefficients	représentation par $n-1$ racines + facteur	représentation par évaluations en n points
évaluation en un point	$O(n)$ (Horner)	$O(n)$ (Horner)	$O(n^2)$ (interpolation)
addition	$O(n)$	?	$O(n)$
multiplication	$O(n^2)$	$O(n)$	$O(n)$

Evaluation en un point avec représentation par coefficients. On utilise la règle de Horner qui évalue le polynôme en un point x_i en $\Theta(n)$:

$$A(x_i) = a_0 + x_i \times (a_1 + x_i \times (a_2 + ... + x_i \times (a_{n-2} + x_i \times a_{n-1}))...))$$

Des trois représentations, on ne va en garder que deux : la *représentation par coefficients* (1.) et de *représentation par valeurs* (3.).

3 Méthodologie

On vient de voir que l'algorithme naïf de multiplication deux polynômes avec la représentation par coefficients est en $\Theta(n^2)$. Mais avec la représentation par valeurs, multiplier deux polynômes est beaucoup plus simple : on multiple terme à terme les valeurs prises. Ainsi, on peut espérer obtenir un algorithme efficace qui passe de la représentation par coefficients dans la représentation par valeurs ; effectue la multiplication rapidement, puis on repasse avec la représentation par coefficients.

Soit $x_0, \ldots, x_{n-1}$ $\underline{n \text{ points}}$ deux à deux distincts. On définit

$$\begin{array}{rccl} & \text{coeffs} & & \text{valeurs} \\ eval : & \mathbb{C}^n & \to & \mathbb{C}^n \\ & (a_0, a_1, \ldots, a_{n-1}) & \mapsto & (A(x_0), \ldots, A(x_{n-1})) \text{ où } A = \sum_{i=0}^{n-1} a_i X^i \end{array}$$

Vision matricielle.
La matrice de *eval* est :

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix}$$

Théorème 6 La fonction *eval* est un isomorphisme d'espaces vectoriels.

L'évaluation *eval* consiste à calculer les valeurs prises par le polynôme aux points $x_0, \ldots, x_{n-1}$, à partir des coefficients de ce polynôme.

Définition 7 (evaluation) L'évaluation est le problème suivant :

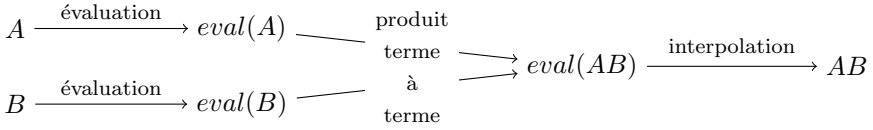
- entrée : $a_0, \ldots, a_{n-1}$
- sortie $eval(a_0, \ldots, a_{n-1})$

L'interpolation $eval^{-1}$ consiste à calculer les coefficient du polynôme à partir des valeurs qu'il prend en $x_0, \ldots, x_{n-1}$:

Définition 8 (interpolation) L'interpolation est le problème suivant :

- entrée : $y_0, \ldots, y_{n-1}$
- sortie : $eval^{-1}(y_0, \ldots, y_{n-1})$.

Pour multiplier deux polynômes A et B à n coefficients, on réalise l'algorithme suivant :



Attention cependant ! Comme AB a $2n-1$ coefficients, il y a besoin d'évaluer en $2n-1$ points ; évaluer en n points seulement ne suffit pas. Dans l'algorithme représenté par le schéma ci-dessus, les évaluations, produit terme à terme, et interpolation se font avec $2n-1$ valeurs à chaque fois. Pour représenter A par coefficients avec $2n-1$ valeurs, il suffit de rajouter des 0. Par exemple si $A = 3X^2 - 2X + 1$, les coefficients sont $1, -2, 3, 0, 0$.

Proposition 9 L'évaluation peut s'effectuer en temps $\Theta(n^2)$.

Démonstration. On évalue en n points et chaque évaluation en un point est en $O(n)$ d'où $O(n^2)$. ■

Proposition 10 L'interpolation peut s'effectuer en temps $\Theta(n^2)$.

Démonstration. On utilise l'interpolation de Lagrange qui est une expression du polynôme à calculer : $\sum_{i=0}^{n-1} y_i \ell_i(X)$ où $\ell_i = \frac{\Pi_{j \neq i}(X - x_j)}{\Pi_{j \neq i}(x_i - x_j)}$. ■

Notre méthodologie ne sert à rien car on est toujours en $\Theta(n^2)$ pour calculer les coefficients de AB. On va maintenant voir comment améliorer la complexité en gardant cette méthodologie.

4 Conception d'un algorithme efficace pour l'évaluation

Si on prend un polynôme pair (avec uniquement des monômes de degré pair), par exemple $A(X) = 42 + X^2 + 3X^4 - X^8$, alors si on connaît l'évaluation $1, 2, 3, 4$, on connaît également l'évaluation en $-1, -2, -3, -4$ car $A(x) = A(-x)$. Si on prend un polynôme impair, par exemple $A(X) = 2X - X^3 + X^9$, alors si on connaît l'évaluation $1, 2, 3, 4$, on connaît également l'évaluation en $-1, -2, -3, -4$ car $A(x) = -A(-x)$.

Idée 1 Au lieu d'évaluer en n points $x_0, x_1, \ldots, x_{n-1}$, on demande à l'ensemble des n points soit clos par négation : il suffit d'évaluer en la moitié des points $x_0, \ldots, x_{n/2-1}$ et on connaît en $-x_0, \ldots, -x_{n/2-1}$.

On décompose le polynôme A en sa partie paire A_{pair}, et impaire A_{impair} : $A(X) = A_{\text{pair}}(X^2) + X \times A_{\text{impair}}(X^2)$.

Exemple 1

A	$=$	1	$+6X$	$-3X^2$	$+2X^3$	$+X^4$	$-7X^5$
A_{pair}	$=$	1		$-3X$		$+X^2$	
A_{impair}	$=$		6		$+2X$		$-7X^2$

On a :

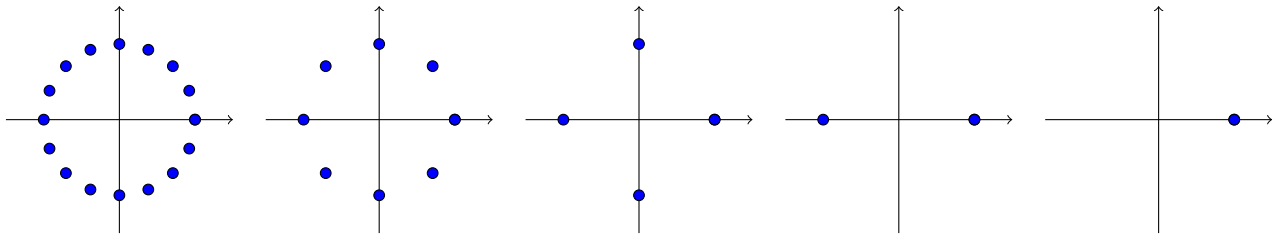
$$A(x_i) = A_{\text{pair}}(x_i^2) + x_i \times A_{\text{impair}}(x_i^2)$$
$$A(-x_i) = A_{\text{pair}}(x_i^2) - x_i \times A_{\text{impair}}(x_i^2)$$

Ainsi, on peut récupérer $A_{\text{pair}}(x_i^2)$ et $A_{\text{impair}}(x_i^2)$ qui servent à la fois pour le calcul de $A(x_i)$ et pour celui de $A(-x_i)$.

Idée 2 On imagine le paradigme diviser pour régner car A_{pair} et A_{impair} sont de taille $n/2$.

Le problème est que dans les deux sous-problèmes, on demande d'évaluer en $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ qui sont a priori tous positifs. Cet ensemble de points n'est pas clos par négatif et on perd donc l'idée 1 qui empêche de répéter l'algorithme récursif.

Idée 3 Si on se place dans $\mathbb{C}$, on peut continuer à avoir des points clos par négation. Pour cela il faut prendre toutes les racines n-ième de l'unité au début. Au deuxième niveau de récursivité, on a les racines $n/2$-ième de l'unité etc.



Dans la suite, on note ω une racine n-ième de l'unité primitive, i.e. qui engendre toutes les autres quand on la met à la puissance.

Bilan. On a de quoi construire un algorithme diviser pour régner efficace pour le problème d'évaluation mais pas en des points quelconques. Il faut évaluer en les racines n-ème de l'unité : $1, \omega, \omega^2, \ldots, \omega^{n-1}$.

5 Transformée de Fourier discrète

Définition 11 (transformée de Fourier discrète) Soit ω une racine n-ème primitive de l'unité. La transformée de Fourier discrète du polynôme $A = \sum_{i=0}^{n-1} a_i X^i$ est le vecteur $DFT_\omega(A) = (A(\omega^0), A(\omega^1), A(\omega^2), \ldots, A(\omega^{n-1}))$. Parfois, on peut noter $DFT_\omega(a)$.

Définition 12 Le problème de calcul de transformée de Fourier Discrète est le problème algorithmique :

- **entrée :**
  - un polynôme A décrit par ses coefficients $a_0, a_1, \ldots, a_{n-1}$;
  - une racine $n^{\text{ème}}$ de l'unité ω ; (par exemple $\omega = e^{\frac{2\pi i}{n}}$ ou $e^{-\frac{2\pi i}{n}}$)
- **sortie :** $DFT_\omega(A)$.

Théorème 13 L'algorithme naïf est en $\Theta(n^2)$.

La transformée de Fourier discrète n'est pas un algorithme, c'est un vecteur. On l'appelle aussi parfois transformation de Fourier discrète, même si la transformation de Fourier désigne généralement l'application *eval* qui à A associe la transformée de Fourier de A. Dans un contexte de traitement de signal, on parle de transformée de Fourier du signal $(a_0, \ldots, a_{n-1})$.

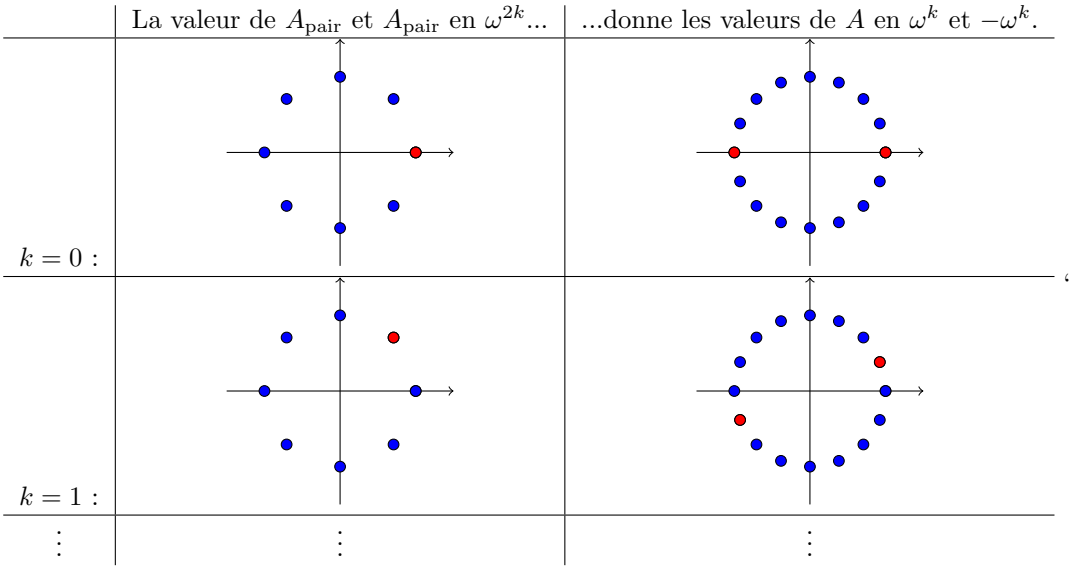
6 Fast Fourier Transform

La FFT (Fast Fourier Transform) est l'algorithme de type diviser pour régner que l'on vient de concevoir. Parfois, par abus de langage, on parle de "la FFT du signal a". Dans cette section, n est une puissance de deux.

6.1 Pseudo-pseudo-code (!) avec l'artillerie des polynômes

Voici le pseudo-code de ce que l'on vient de voir tout à l'heure :

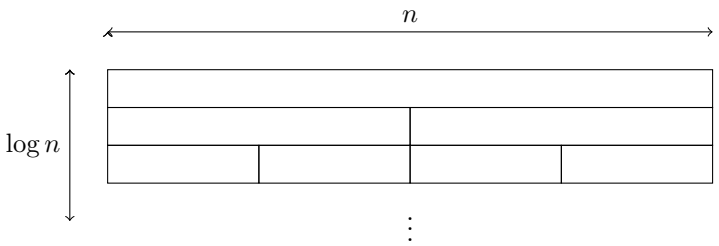
entrée : un polynôme A décrit par ses coefficients $(a_0, \dots, a_{n-1})$, une racine de n-ème primitive de l'unité ω
sortie : $A(\omega^0), A(\omega^1), A(\omega^2), \dots, A(\omega^{n-1})$
fonction FFT(A, ω)
  si A est constant
    renvoyer $A(1)$
  sinon
    soit A_{pair} la partie paire et A_{impair} la partie impaire de A
    calculer $A_{\text{pair}}(1)$, $A_{\text{pair}}(\omega^2)$ $\dots$, $A_{\text{pair}}(\omega^{n-2})$ via $FFT(A_{\text{pair}}, \omega^2)$
    calculer $A_{\text{impair}}(1)$, $A_{\text{impair}}(\omega^2)$ $\dots$, $A_{\text{impair}}(\omega^{n-2})$ via $FFT(A_{\text{impair}}, \omega^2)$
    pour $k = 0$ à $n/2 - 1$
      $A(\omega^k) := A_{\text{pair}}(\omega^{2k}) + \omega^k A_{\text{impair}}(\omega^{2k})$
      $A(-\omega^k) := A_{\text{pair}}(\omega^{2k}) - \omega^k A_{\text{impair}}(\omega^{2k})$
  renvoyer $A(\omega^0), A(\omega^1), A(\omega^2), \dots, A(\omega^{n-1})$



Théorème 14 La complexité est en $\Theta(n \log n)$.

Démonstration.

$$\tau(n) = 2\tau\left(\lceil \frac{n}{2} \rceil\right) + \Theta(n).$$



■

6.2 Pseudo-code sans l'artillerie des polynômes

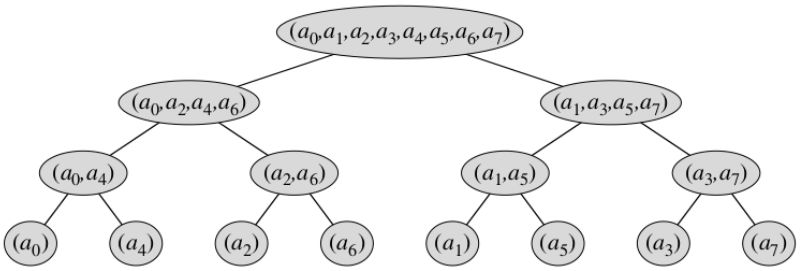
entrée : des nombres $a_0, \ldots, a_{n-1}$, et une racine n-ème de l'unité primitive ω
sortie : $A(\omega^0), A(\omega^1), A(\omega^2), \ldots, A(\omega^{n-1})$ où $A = \sum_{i=0}^{n-1} a_i X^i$

fonction FFT$(a_0, \ldots, a_{n-1},\ \omega)$
  si $n = 1$
    renvoyer a_0
  sinon
    $(s_0, \ldots, s_{n/2-1}) := FFT(a_0, a_2, \ldots, a_{n-2},\ \omega^2)$
    $(s'_0, \ldots, s'_{n/2-1}) := FFT(a_1, a_3, \ldots, a_{n-1},\ \omega^2)$
    $\alpha := 1$
    pour $k = 0$ à $n/2 - 1$
      $y_k := s_k + \alpha s'_k$
      $y_{k+n/2} := s_k - \alpha s'_k$
      $\alpha := \alpha \times \omega$
  renvoyer $y_0, \ldots, y_{n-1}$

7 Algorithme de la FFT en place

On peut implémenter l'algorithme de la FFT en place, c'est-à-dire sans devoir allouer des nouveaux tableaux à chaque fois. De plus, on va voir que l'on peut exécuter des instructions en parallèles.

Voici l'arbre des appels récursifs de la FFT :

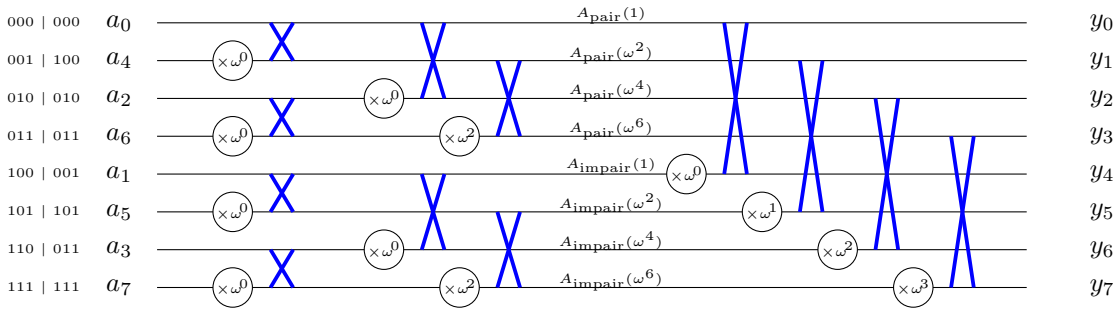


À la racine, on met les coefficients dont le numéro est pair à gauche, et les coefficients dont le numéro est impair. Autrement dit, les coefficients dont l'indice est de poids faible = 0 sont en premier, alors que les autres, de poids faible = 1 sont en dernier. Ainsi les coefficients de la première partie sont paires et, comme ils sont au début, leur nouvel indice est de la forme 0..... Ceux d'indices impairs, comme ils sont à la fin, auront un nouvel indice de la forme 1.....

Le processus continue et du coup cela revient à réécrire les indices en prenant le miroir des écritures binaires. L'ordre 0, 4, 2, 6, 1, 5, 3, 7 s'obtient en prenant les miroirs des écritures binaires de 0, 1, 2, 3, 4, 5, 6, 7 (bit-reversal).

Le circuit suivant représente le calcul de la FFT, en lisant l'arbre ci-dessous. Les feuilles sont les entrées.

Le calcul en bleu dans l'algorithme s'appelle une *opération papillon* : on la représente par une croix bleue, agrémentée par la valeur de α.



Théorème 15 Il y a $\Theta(n \log n)$ opération papillons.

Théorème 16 Le circuit est de profondeur $\Theta(\log n)$. L'algorithme s'exécute en $\Theta(\log n)$ sur une architecture parallèle.

7.1 Algorithme itératif

entrée : des nombres $a_0, \ldots, a_{n-1}$, et une racine n-ème de l'unité primitive ω
sortie : $A(\omega^0), A(\omega^1), A(\omega^2), \ldots, A(\omega^{n-1})$ où $A = \sum_{i=0}^{n-1} a_i X^i$

fonction FFT$(a_0, \ldots, a_{n-1},\ \omega)$
  pour $k = 0$ à $n-1$
    $s_k := a_{bitreversal(k)}$
  pour $m := 2, 4, 8, \ldots, n$
    pour $k = 0, m, 2m, \ldots, n-m$
      $\alpha := 1$
      pour $j = 0$ à $m/2 - 1$

$$\begin{pmatrix} s_{k+j} \\ s_{k+j+m/2} \end{pmatrix} := \begin{pmatrix} s_{k+j} + \alpha s_{k+j+m/2} \\ s_{k+j} - \alpha s_{k+j+m/2} \end{pmatrix}$$

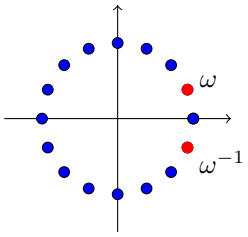
        $\alpha := \alpha \times \omega^{n/m}$
  renvoyer $s_0, \ldots, s_{n-1}$

8 Et l'interpolation alors ?

On vient de voir que la $DFT_\omega(a)$ permet de calculer l'évaluation du polynôme en $\omega^0, \omega^1, \ldots, \omega^{n-1}$ dont les coefficients sont a en $\Theta(n \log n)$ où a est de taille n. La chose géniale est que l'*interpolation*, c'est-à-dire le calcul des coefficients à partir des valeurs prises en $\omega^0, \omega^1, \ldots, \omega^{n-1}$ se fait... avec le même algorithme !

Proposition 17 On considère les points $x_0, \ldots, x_{n-1}$ qui sont $\omega^0, \omega^1, \ldots, \omega^{n-1}$. On a :

- $eval = DFT_\omega$; (évaluation)
- $eval^{-1} = \frac{1}{n} DFT_{\omega^{-1}}$. (interpolation)



Démonstration. La démonstration repose sur l'étude de la matrice de Van Der Monde suivante :

$$V(\omega) = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\times 2} & \ldots & \omega^{2\times(n-1)} \\ \vdots & \ldots & \ldots & \ldots & \vdots \\ 1 & \omega^{n-1} & \omega^{(n-1)\times 2} & \ldots & \omega^{(n-1)(n-1)} \end{pmatrix}.$$

On a $DFT_\omega(a) = V(\omega)a$.
On montre que

$$V(\omega^{-1}) = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \ldots & \omega^{-n+1} \\ 1 & \omega^{-2} & \omega^{-2\times 2} & \ldots & \omega^{-2\times(n-1)} \\ \vdots & \ldots & \ldots & \ldots & \vdots \\ 1 & \omega^{1-n} & \omega^{-(n-1)\times 2} & \ldots & \omega^{-(n-1)(n-1)} \end{pmatrix}.$$

On montre que $V(\omega) \times V(\omega^{-1}) = nI_n$. Donc $V(\omega)$ est inversible et son inverse est la matrice $\frac{1}{n}V(\omega^{-1})$. La matrice de l'application linéaire $eval^{-1}$ est $\frac{1}{n}V(\omega^{-1})$. Autrement dit, $eval^{-1} = \frac{1}{n}DFT_{\omega^{-1}}$. ■

9 Notes bibliographiques

Pour la FFT, on cite souvent Cooley et Tukey [CT65], mais l'algorithme était connu et on attribue sa première invention à Gauss. Elle est décrite dans [DPV08] et [CLRS09].

References

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

ALGO1 – Tables de hachage

François Schwarzentruber

1 Exemple - Quizz

- entrée : un tableau de n nombres
- sortie : oui s'il existe deux nombres a et b dans un tableau T tel que $a + b = 0$.

Il y a un algorithme naïf en $O(n^2)$. Mais on peut faire mieux avec un algorithme de cette forme :

```
fonction algo(T)
    S := construire un ensemble contenant les éléments de T
    pour a ∈ S
        si −a ∈ S
            renvoyer vrai
    renvoyer faux
```

Selon l'efficacité de la structure de données utilisée, on peut avoir une complexité meilleure.

2 But

Le but de ce cours est de discuter une structure de données pour implémenter les types abstraits ensemble et tableau associatif (dictionnaire).

Ensemble = ensemble fini d'éléments

- ajouter/supprimer un élément
- tester l'appartenance
- union etc.

Dictionnaire = fonction fini associant des valeurs à des clefs :

- ajouter/remplacer une clé-valeur $T[clef] := valeur$
- avoir la valeur $T[clef]$

Relation symétrique Tester si une relation donnée sous la forme d'une liste de paires (a, b) est symétrique.

Texte : mots les plus fréquents Étant donné un roman, quels sont les 100 mots les plus fréquents ? Utilisé ensuite par exemple pour comparer deux textes

Base de données On a une liste de couples $(prixapayer, client)$. On souhaite en temps linéaire calculer pour chaque client le prix total qu'il doit payer.

Dans un compilateur/interpréteur Pour stocker l'association entre nom de variables et données.

Routeur Pour trouver quel ordinateur contacter si on veut les données d'une certaine adresse.

3 Définition

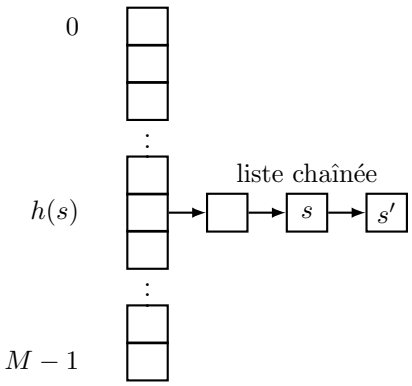
L'idée est d'utiliser les indices d'un tableau comme élément. $T[e] := 1$ revient à ajouter e à l'ensemble. Cette structure de données s'appelle *vecteur de bits*.

Le souci est que les éléments peuvent être quelconque : des nombres très grands, des chaînes de caractères etc. Donc on ne peut pas utilisé les éléments directement. On introduit donc une fonction de hachage.

Définition 1 (fonction de hachage) Une fonction de hachage est une fonction $h : \mathbb{K} \to [0, M-1]$.

Faire l'exemple au tableaunoir.github.io avec des objets qu'on met dans des boîtes.

Définition 2 (table de hachage par chaînage) Une table de hachage par chaînage est une structure de données composée d'un tableau avec M cases, chacune contenant une liste chaînée.




```
fonction contient(s)
    renvoyer liste_contient(T[h(s)], s)
```

```
fonction ajouter(s)
    renvoyer liste_ajouter(T[h(s)], s)
```

```
fonction supprimer(s)
    renvoyer liste_supprimer(T[h(s)], s)
```

Définition 3 (alvéole) Une case du tableau s'appelle une alvéole.

Définition 4 (collision) Une collision est un ensemble de 2 clefs logées dans la même alvéole.

Notations

Ω	= l'univers des mondes possibles (pour parler de probabilités)
$\mathbb{K}$	= l'univers des clefs possibles (pas forcément dans la table)
n	= nombre de clefs présentes dans la table
M	= nombre d'alvéoles dans la table
h	= fonction de hachage
a_j	= nombre de clefs dans l'alvéole numéro j
$\alpha = \frac{n}{M}$	= facteur de remplissage

4 Hachage uniforme

Définition 5 (Hypothèse du hachage uniforme simple) Soit $K : \Omega \to \mathbb{K}$ est une variable aléatoire représentant une clef. La clef K vérifie l'hypothèse de hachage uniforme simple si pour tout alvéole $j \in \{0, \dots, M-1\}$,

$$\mathbb{P}(h(K) = j) = \frac{1}{M}.$$

demo: Hachage uniforme

Dans ce cas là, on a la proposition suivante, avec $\leq$ car les éléments qu'on ajoute peuvent être égaux et il n'y a pas de doublons :

Proposition 6 Supposons que l'on ait ajouté n éléments : $K_1, \ldots, K_n : \Omega \to \mathbb{K}$ sous l'hypothèse de hachage uniforme simple. On a alors pour tout $j \in \{0, \ldots, M-1\}$,

$$\mathbb{E}(a_j) \leq \alpha = \frac{n}{M}.$$

Démonstration.

Notation 7 (fonction indicatrice) Soit $\mathcal{P}$ une propriété. On note $1_{\mathcal{P}}$ la fonction de $\Omega \to \{0,1\}$ qui vaut 1 si la propriété $\mathcal{P}$ est vraie et 0 sinon.

Le nombre de tentatives d'ajout d'une clef dans l'alvéole numéro j vaut $\sum_{i=1}^{n} 1_{h(K_i)=j}$. On a donc :

$$a_j \leq \sum_{i=1}^{n} 1_{h(K_i)=j}.$$

Le $\leq$ provient du fait qu'il peut y avoir des clefs égales. On a :

$$\begin{array}{lll} \mathbb{E}(a_j) & \leq \mathbb{E}(\sum_{i=1}^{n} 1_{h(K_i)=j}) & \\ & = \sum_{i=1}^{n} \mathbb{E}(1_{h(K_i)=j}) & \text{par linéarité de l'espérance} \\ & = \sum_{i=1}^{n} \mathbb{P}(h(K_i)=j) & \\ & = \sum_{i=1}^{n} \frac{1}{M} & \text{par hypothèse du hachage uniforme simple} \\ & = \frac{n}{M} & \end{array}$$

■

Corollaire 8 Les opérations d'ajout, suppression et recherche sont $O(1+\alpha)$ en moyenne.

5 Hachage universel

Problème : Un intrus qui connaît la fonction de hachage peut attaquer la table de hachage.
demo: Hachage uniforme : attaque
Solution : choisir aléatoirement une fonction de hachage H.

Exemple 9 Soit p un nombre premier. Supposons que l'ensemble des clefs soit $\mathbb{K} = \mathbb{Z}/_{p\mathbb{Z}}$.
On pose $\mathcal{D} := \mathbb{Z}/^{*}_{p\mathbb{Z}} \times \mathbb{Z}/_{p\mathbb{Z}}$.
Tirer aléatoirement uniformément (a,b) dans $\mathcal{D}$ et choisir comme fonction de hachage

$$\begin{array}{rccl} h_{ab} : & \mathbb{Z}/_{p\mathbb{Z}} & \to & \{0, \ldots, M-1\} \\ & k & \mapsto & ((ak+b)[p])[M]. \end{array}$$

TODO: lire une intuition sur ça

demo: Hachage universel

5.1 Fonction de hachage aléatoire universelle

Définition 10 (fonction de hachage aléatoire universelle) Soit $H : \Omega \to (\mathbb{K} \to \{1, \ldots, M\})$ une fonction de hachage aléatoire est universelle si pour toute paire de clef $(k, \ell) \in \mathbb{K}$ tel que $k \neq \ell$, on a

$$\mathbb{P}(H(k) = H(\ell)) \leq \frac{1}{M}.$$

Théorème 11 Supposons que H soit une fonction de hachage aléatoire universelle. Alors pour tous éléments $x_1, \ldots, x_n$ distincts deux à deux, pour tout x, après l'insertion de $x_1, \ldots, x_n$ on a :

$$\mathbb{E}(a_{H(x)}) \leq \frac{n}{M} + 1.$$

Démonstration.

$$\begin{array}{lll} \mathbb{E}(a_{H(x)}) & = \mathbb{E}(\sum_{i=1}^{n} 1_{H(x_i)=H(x)}) & \text{par définition de } a_{H(x)} \\ & = \sum_{i=1}^{n} \mathbb{E}(1_{H(x_i)=H(x)}) & \text{par linéarité de l'espérance} \\ & = \sum_{i=1}^{n} \mathbb{P}(H(x_i) = H(x)) & \end{array}$$

Si $x_i = x$, $\mathbb{P}(H(x_i) = H(x)) = 1$. Si $x_i \neq x$, $\mathbb{P}(H(x_i) = H(x)) \leq \frac{1}{M}$.
D'où le résultat. ■

5.2 L'exemple est une classe universelle (*)

Cette section peut ne pas être traitée en cours car c'est assez pénible. Dans cette section, on considère un nombre premier p qui est plus grand que l'univers des clefs possibles, de sorte que l'on prend $\mathbb{K} = \mathbb{Z}/_{p\mathbb{Z}}$.

Théorème 12 (admis) Soit $\mathcal{D} = \mathbb{Z}/^{*}_{p\mathbb{Z}} \times \mathbb{Z}/_{p\mathbb{Z}}$. Si on tire aléatoirement uniformément (a, b) dans $\mathcal{D}$, alors la variable aléatoire h_{ab} de l'exemple 8 est universelle.

Démonstration. Rappel :

$$\begin{array}{rccl} h_{ab}: & \mathbb{Z}/_{p\mathbb{Z}} & \to & \{0, \ldots, M-1\} \\ & k & \mapsto & (\underbrace{(ak+b)[p]}_{f_{ab}(k)})[M]. \end{array}$$

Par définition, montrer que h_{ab} est universelle revient à montrer que pour toute paire de clef (k, ℓ) tel que $k \neq \ell$, on a

$$|\{(a, b) \in \mathcal{D} \mid h_{ab}(k) = h_{ab}(\ell)\}| \leq \frac{|\mathcal{D}|}{M}.$$

Il faut montrer que pour toute paire de clef (k, ℓ) tel que $k \neq \ell$, on a

$$|\{(a, b) \in \mathcal{D} \mid f_{ab}(k) = f_{ab}(\ell)[M]\}| \leq \frac{|\mathcal{D}|}{M}.$$

Soit $\mathcal{S} = \{(r, s) \in \mathbb{Z}/^{2}_{p\mathbb{Z}} \mid r \neq s\}$. Soit $k, \ell \in U$ telles que $k \neq \ell$. Posons

$$\begin{array}{rccl} \theta: & \mathcal{D} & \to & \mathcal{S} \\ & (a, b) & \mapsto & (f_{ab}(k), f_{ab}(\ell)) \end{array}$$

Lemme 13 θ est une bijection.

Démonstration. Bien défini θ est bien à valeurs dans $\mathcal{S}$. En effet :

- D'une part, $f_{ab}(k), f_{ab}(\ell) \in \mathbb{Z}/_{p\mathbb{Z}}$ par définition de f_{ab}.
- D'autre part $f_{ab}(k) \neq f_{ab}(\ell)$. En effet, par définition de $f_{ab}(..)$, il faut montrer que $ak + b \neq a\ell + b[p]$. Autrement dit, il faut montrer que $a(k - \ell) \neq 0[p]$. Mais :
  - $a \neq 0$ (par définition de $\mathcal{D}$).
  - Et comme p est plus grand que les valeurs des clefs (ie. $k < p$ et $\ell < p$), et $k - \ell \neq 0$ implique que $(k - \ell) \neq 0[p]$.

  Comme $\mathbb{Z}/_{p\mathbb{Z}}$ est intègre, on a bien $a(k - \ell) \neq 0[p]$. CQFD.

Surjection

Pour tout $(r,s) \in \mathcal{S}$, montrons qu'il existe $(a,b) \in \mathcal{D}$ tel que $\theta(a,b) = (r,s)$. Le candidat (a,b) est :

- $a = (r-s)(k-\ell)^{-1}[p]$;
- $b = (r - ak)[p]$.

On remarque que comme $k \neq \ell$ et comme $r \neq s$, alors par intégrité de $\mathbb{Z}/_{p\mathbb{Z}}$, $a \neq 0[p]$. Donc $(a,b) \in \mathcal{D}$. D'autre part,

$$\begin{aligned}
\theta(a,b) &= (ak+b, a\ell+b) \\
&= (ak + \underbrace{(r-ak)}_{b}, \quad \underbrace{(r-s)(k-\ell)^{-1}}_{a}\ell + \underbrace{r - \underbrace{(r-s)(k-\ell)^{-1}}_{a}k}_{b}) \\
&= (r, \quad (r-s)(k-\ell)^{-1}(\ell-k)+r) \\
&= (r, \quad s-r+r) \\
&= (r,s).
\end{aligned}$$

Conclusion

Comme $|\mathcal{D}| = |\mathcal{S}|$, on en conclut que θ est une bijection.

■

Vu que θ est une bijection, on a :

$$|\{(a,b) \in \mathcal{D} \mid f_{ab}(k) = f_{ab}(\ell)[M]\}| = |\{(r,s) \in \mathcal{S} \mid r = s[M]\}|.$$

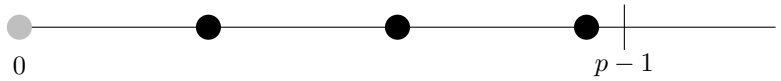
Ainsi, l'objectif est de montrer que :

$$|\{(r,s) \in \mathcal{S} \mid r = s[M]\}| \leq \frac{|\mathcal{S}|}{M}.$$

Pour un $r \in \mathbb{Z}/_{p\mathbb{Z}}$ fixé,
$|\{s \in \mathbb{Z}/_{p\mathbb{Z}} \mid (r,s) \in \mathcal{S} \text{ et } r = s[M]\}|$

$$\begin{aligned}
&= |\{s \in \mathbb{Z}/_{p\mathbb{Z}} \mid (r,s) \in \mathbb{Z}/^2_{p\mathbb{Z}} \text{ et } r \neq s[p] \text{ et } r = s[M]\}| \\
&\leq \left\lceil \tfrac{p}{M} \right\rceil - 1 \\
&\leq \tfrac{p+M-1}{M} - 1 = \tfrac{p-1}{M}
\end{aligned}$$

Exemple 14 Par exemple pour $r = 0$, voici les candidats (en noir) espacés de M :



Ainsi,

$$|\{(r,s) \in \mathcal{S} \mid r = s[M]\}| \leq p\frac{p-1}{M} = \frac{|\mathcal{S}|}{M}$$

■

5.3 Autre exemple d'une classe universelle

On considère ici un exemple dont la démonstration d'universalité est plus simple. Par contre, c'est au prix d'un ensemble de départ et une taille M de tableau un peu tarabiscotés. Soit p un nombre premier. On suppose que l'univers des clefs est $\mathbb{K} = \{0,\ldots,p-1\}^r$ où r est un entier. Pour cela, on suppose que chaque clef x qui est une suite de bits est décomposé en $(x_1,\ldots,x_r)$ où chaque $x_i \in \{0,\ldots,p-1\}$.

Pour choisir une fonction de hachage, on procède comme suit :

1. on choisit uniformément $a = (a_1, \ldots, a_r) \in \{0, \ldots, p-1\}^r$;
2. on prend $\begin{array}{cccc} & \mathbb{K} & \to & \mathbb{Z}/_{p\mathbb{Z}} \\ h_a & x & \mapsto & \sum_{i=1}^{r} a_i x_i [p] \end{array}$ comme fonction de hachage.

On considère ici que $M = p$.

Théorème 15 La variable aléatoire h_a est universelle.

DÉMONSTRATION. Soit $x \neq y$ deux clefs différentes. Montrons que $\mathbb{P}(h_a(x) = h_a(y)) \leq \frac{1}{p}$. Comme $x \neq y$, il existe un indice $j \in \{1, \ldots, r\}$ tel que $x_j \neq y_j$.

$$\begin{aligned} h_a(x) = h_a(y) \text{ ssi } & \sum_{i=1}^{r} a_i x_i = \sum_{i=1}^{r} a_i y_i [p] \\ \text{ssi } & a_j \underbrace{(y_j - x_j)}_{z} = \underbrace{\sum_{i \neq j} a_i (x_i - y_i)[p]}_{m} \end{aligned}$$

On a $z \neq 0$. L'équation $a_j z = m[p]$ admet une seule solution a_j dans $\{0, \ldots, p-1\}$. C'est parce que $\mathbb{Z}/_{p\mathbb{Z}}$ est un corps, la solution s'écrit $a_j = mz^{-1}$.
Donc $\mathbb{P}(h_a(x) = h_a(y)) = \frac{1}{p}$.
■

On a supposé que $M = p$. Ce n'est grave à constante près, car il y a suffisamment de nombres premiers. C'est le postulat de Bertrand qui dit que pour tout M, il existe un nombre premier entre M et $2M$.

5.4 Nombre de collisions

Lemme 16 Soit H une fonction de hachage aléatoire avec l'hypothèse d'universalité. On a :

$$\mathbb{E}(\text{nb de collisions obtenues avec } H) \leq \binom{n}{2} \frac{1}{M}.$$

DÉMONSTRATION. Notons X le nombre de collisions obtenues avec H. On a :

$$X = \sum_{\{k,\ell\} \text{ où } k,\ell \text{ dans la table } | k \neq \ell} 1_{H(k)=H(\ell)}.$$

En effet, on compte un pour chaque ensemble $\{, \ell\}$ de clefs distinctes quand elle forme une collision, i.e. quand $H(k) = H(\ell)$. Par linéarité de l'espérance on a :

$$\mathbb{E}(X) = \sum_{\{k,\ell\} \text{ où } k,\ell \text{ dans la table } | k \neq \ell} \mathbb{E}(1_{H(k)=H(\ell)})$$

Mais

$$\begin{array}{ll} \mathbb{E}(1_{H(k)=H(\ell)}) & = \mathbb{P}(H(k) = H(\ell)) \\ & \leq \frac{1}{M} \qquad \text{car } H \text{ est universelle.} \end{array}$$

On a :

$$\mathbb{E}(X) = \sum_{\{k,\ell\} \text{ où } k,\ell \text{ dans la table } | k \neq \ell} \mathbb{E}(1_{H(k)=H(\ell)}) \leq \sum_{\{k,\ell\} \text{ où } k,\ell \text{ dans la table } | k \neq \ell} \frac{1}{M} \leq \binom{n}{2} \frac{1}{M}.$$

■

6 Hachage parfait

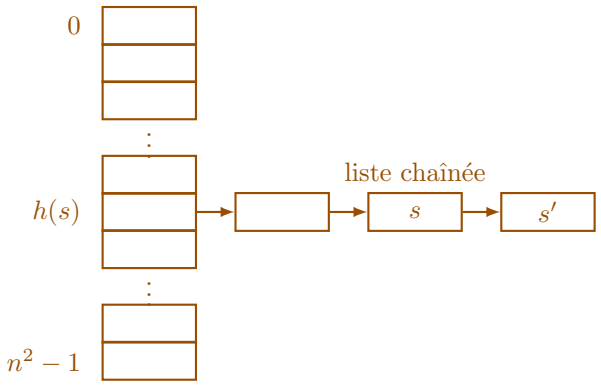
Problème : les complexités sont en moyenne mais pas dans le pire cas.
Solution : étant donné n clés **fixées et connues**, *compiler* une structure de données pour avoir un accès pire cas en $O(1)$ et un espace mémoire en $O(n)$

Exemple d'outil : gperf

Utilisation dans un parser (V8 par exemple)

6.1 Étape 1 : table avec accès $O(1)$ et taille $O(n^2)$

demo: Hachage parfait



Théorème 17 Posons $M = n^2$. Soit H une fonction de hachage choisie aléatoire l'hypothèse d'universalité. Alors :

$$\mathbb{P}(\text{pas de collision avec } H) > \frac{1}{2}.$$

Démonstration. Soit X le nombre de collisions que l'on a avec H. Via le lemme 13, on a :

$$\mathbb{E}(X) \leq \frac{n(n-1)}{2}\frac{1}{n^2} < \frac{1}{2}.$$

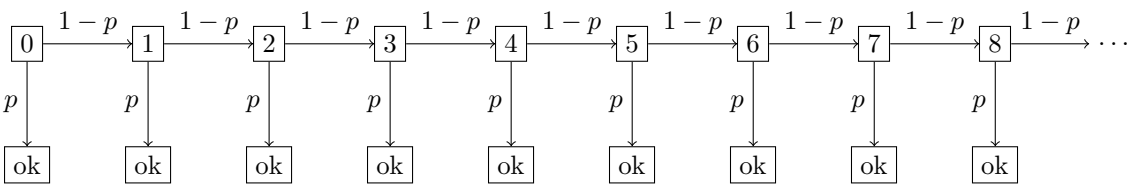
Rappel de l'inégalité de Markov : $\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}(X)}{t}$.

Ici, pour $t = 1$ l'inégalité de Markov donne $\mathbb{P}(X \geq 1) \leq \frac{1}{2}$. Donc $\mathbb{P}(\text{non } X \geq 1) = \mathbb{P}(X = 0) > \frac{1}{2}$. Autrement dit, $\mathbb{P}(\text{pas de collision avec } H) > \frac{1}{2}$. ■

Solution : tirer aléatoirement h dans $\mathcal{H}$ et s'arrêter quand on n'a pas de collision.

Théorème 18 Il faut en moyenne $\frac{1}{p}$ essais pour avoir une fonction de hachage qui ne donne pas de collisions, où p est la probabilité de ne pas avoir de collisions.

Démonstration.
Soit T le nombre d'essais jusqu'à ne pas avoir de collision (essais ratés puis l'essai final qui réussit).



$\mathbb{P}(T=0)=0$
$\mathbb{P}(T=1)=p$
$\vdots$
$\mathbb{P}(T=k)=(1-p)^{k-1}p$
$\vdots$
T suit une *loi géométrique* de paramètre p.
Il faut calculer $\mathbb{E}(T)$. On a :

$$\begin{aligned}
\mathbb{E}(T) &= \sum_{1}^{\infty} k\mathbb{P}(T=k) && \text{définition de l'espérance}\\
&= \sum_{0}^{\infty} k(1-p)^{k-1}p && \text{en remplaçant par la valeur de } \mathbb{P}(T=k) \text{ (voir plus bas)}\\
&= p\frac{1}{(1-(1-p))^2} && \text{car on reconnaît une série convergente (voir plus bas)}\\
&= \frac{1}{p} && \text{simplification mathématique}
\end{aligned}$$

.

Comme $\sum_0^\infty x^k = \frac{1}{1-x}$, on a $\sum_1^\infty kx^{k-1} := \frac{1}{(1-x)^2}$.

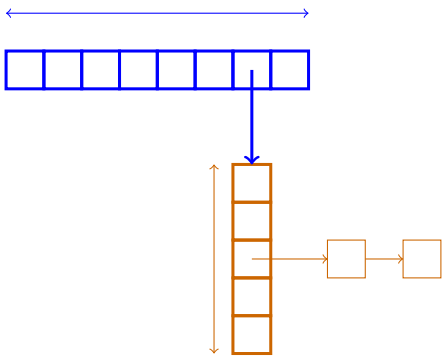
La fonction $p \mapsto \frac{1}{p}$ est décroissante en p et vaut 2 en $\frac{1}{2}$. Il faut donc en moyenne "moins de 2 tirages" pour ne pas avoir de collisions. ■

6.2 Étape 2 : obtenir une taille en $O(n)$

On construit une structure à deux niveaux :

1. Premier niveau avec $M=n$ alvéoles, avec une fonction de hachage h. Il peut y avoir des collisions.
2. Dans l'alvéole numéro j du premier niveau, on met une table de hachage sans collision T_j. On utilise l'étape 1 : on met ${a_j}^2$ sous-alvéoles dans T_j où a_j = le nombre d'éléments dans l'alvéole numéro j.

   On montre que l'on peut choisir h du premier niveau tel que l'espace du deuxième niveau $\sum_{j=0}^{M-1} {a_j}^2 = O(n)$.



Théorème 19 Posons $M=n$. Soit H une fonction de hachage aléatoire universelle. Soit ${a_j}^H$ la variable aléatoire qui représente le nombre d'éléments dans l'alvéole numéro j au premier niveau avec H. Alors :

$$\mathbb{E}(\sum_{j=0}^{M-1} \left({a_j}^H\right)^2) < 2n.$$

Démonstration. On va calculer $\sum_{j=0}^{M-1} \left(a_j{}^H\right)^2$.

Rappel : on a l'égalité pour tout $a \in \mathbb{N}$: $a^2 = a + 2\binom{a}{2}$.

On a :

$$\sum_{j=0}^{M-1} \left(a_j{}^H\right)^2 = \underbrace{\sum_{j=0}^{M-1} a_j{}^H}_{n} + 2 \underbrace{\sum_{j=0}^{M-1} \binom{a_j{}^H}{2}}_{X}$$

En fait, X est le nombre de collisions dans la table de hachage principale. En effet, $\binom{a_j{}^H}{2}$ est le nombre de couples de clefs $\{k, \ell\}$ avec $k \neq \ell$ où $H(k) = H(\ell) = j$. En faisant la somme, on compte toutes les collisions.

Ainsi, on a :

$$\begin{array}{lll} \mathbb{E}(\sum_{j=0}^{M-1} a_j{}^{H\,2}) & \leq n + 2\mathbb{E}(X) & \text{par la linéairité de l'espérance} \\ & \leq n + 2\binom{n}{2}\frac{1}{n} & \text{par le lemme 13} \\ & \leq n + 2\frac{n-1}{2} & \\ & < 2n. & \end{array}$$

■

6.3 Construction de la structure

1. On tire au hasard h pour le premier niveau jusqu'à avoir $\sum_{j=0}^{M-1} a_j{}^2 < 4n$.

   La probabilité d'avoir $\sum_{j=0}^{M-1} a_j{}^2 < 4n$ est minoré par

   $$\begin{array}{lll} \mathbb{P}(\sum_{j=0}^{M-1} a_j{}^2 < 4n) & = 1 - \mathbb{P}(\sum_{j=0}^{M-1} a_j{}^2 \geq 4n) & \\ & \geq 1 - \frac{\mathbb{E}(\sum_{j=0}^{M-1} a_j{}^{H\,2})}{4n} & \text{inégalité de Markov} \\ & > 1 - \frac{2n}{4n} & \\ & > \frac{1}{2}. & \end{array}$$

2. Pour tout j, pour la table de hachage secondaire dans l'alvéole numéro j, on choisit une fonction de hachage h_j qui ne donne pas de collision (cf. étape 1).

7 FAQ

Où est-ce que le hachage parfait est-il utilisé ? Quand les données sont fixés une fois pour toute. Un exemple typique est dans un compilateur où un dictionnaire mot clefs $\mapsto$ action à prendre est stockés avec du hachage parfait. Le dictionaire d'une langue est un autre exemple.

8 Notes bibliographiques

Cette note de cours résume le chapitre sur les tables de hachage de [CLRS09]. A la fin du chapitre, ils indiquent que Donald Knuth attribue l'invention des tables de hachage à Hans Peter Luhn. Le hachage parfait avec des clefs statiques a été proposé dans [FKS84]. Le cas dynamique, avec des ajouts et suppressions en $O(1)$ en cout amorti est proposé dans [DKM+94].

References

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[DKM+94] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

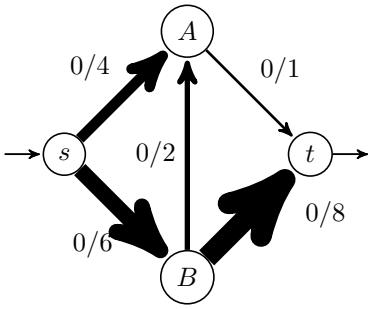
[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

Flots

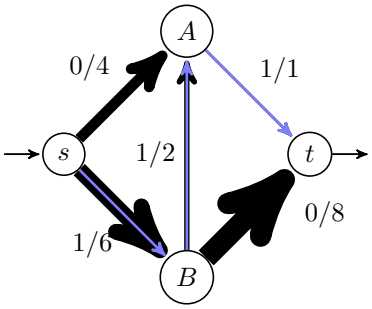
François Schwarzentruber

5 octobre 2023

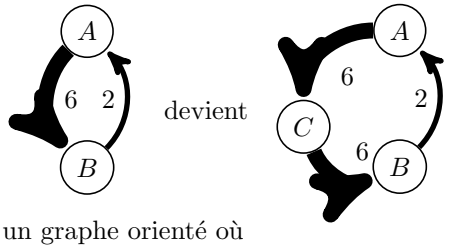
1 Définitions



Un réseau de flot est un graphe orienté, pondéré positivement par des entiers. On distingue deux sommets : une source s et un puits t. On y fait circuler de l'eau de la source s au puits t.



Pour simplifier notre discours, on suppose sans perte de généralité qu'il n'y a pas d'arcs *anti-parallèles* dans le graphe, c'est-à-dire il n'y a jamais d'arcs de A vers B et un de B vers A. On peut toujours supprimer les arcs anti-parallèles.



Définition 1 (réseau de flot) Un réseau de flot $G = (V, E, c, s, t)$ est un graphe orienté où
- (V, E) est un graphe orienté irréflexif, sans arcs anti-parallèles ;
- $c : E \to \mathbb{R}^*$;
- s est un sommet appelé source, sans arcs entrants ;
- t est un sommet appelé puits, sans arcs sortants.

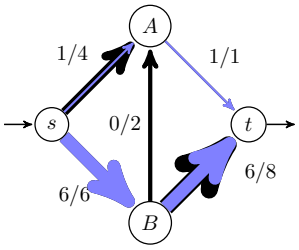
On note $c(u, v)$ la capacité de l'arête (u, v). Par convention, si l'arête (u, v) n'existe pas, $c(u, v) = 0$.

Définition 2 (flot) Un flot de $G = (V, E)$ est une fonction $f : V^2 \to \mathbb{R}^+$ telle que :
1. Pour tous sommets u, v dans V, on a $0 \leqslant f(u, v) \leqslant c(u, v)$.
2. Pour tout sommet $\underline{u}$ de V différent de s et de t, $\sum_{v \in V} f(v, \underline{u}) = \sum_{w \in V} f(\underline{u}, w)$. (loi de Kirchhoff)

Définition 3 (valeur du flot) La valeur du flot f est $|f| = \sum_{v \in V} f(s, v)$.

Définition 4 (flot maximal) Un flot f est maximal de G si la valeur $|f|$ est maximale,
i.e. pour tout flot g de G, $|g| \leq |f|$.

Exemple 1 Voici un flot maximum de valeur 7 :

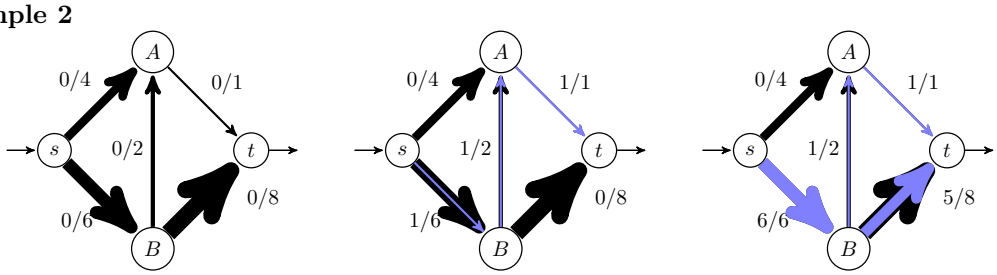


Définition 5 (problème du flot maximum)
- entrée : un réseau G ;
- sortie : un flot maximal de G.

2 Échec d'une méthode gloutonne

pre : Un réseau G
post : l'algorithme ne garantit pas d'avoir un flot f maximal
fonction algoMalFichu(G)
    f := flot nul
    tant que il existe un chemin simple de s à t qui améliore le flot
        augmenter f à l'aide de ce chemin
    renvoyer f

Exemple 2



On obtient un flot de 6 et là on est bloqué, jamais on ne trouvera un flot de 7.

3 Graphe résiduel

Comme le montre l'exemple précédent, on voudrait annuler les mauvais choix. content...

Exemple 3 On voudrait annuler l'eau passant par l'arrête $B \to A$ et la rediriger de B à t.

On introduit le graphe résiduel qui indique les modifications possibles du flot courant :
— Ajout d'eau ;
— Annulation d'eau.

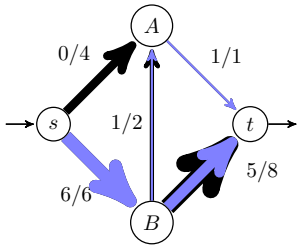
Définition 6 (graphe résiduel) Le graphe résiduel d'un réseau $G = (V, E, c, s, t)$ et d'un flot f est le graphe pondéré $R_f = (V, A_f, r_f)$ où les sommets sont V, la fonction de poids r_f est :

$$r_f(u,v) = \begin{cases} c(u,v) - f(u,v) \text{ si } (u,v) \in E \\ f(v,u) \text{ si } (v,u) \in E \\ 0 \text{ sinon} \end{cases}$$

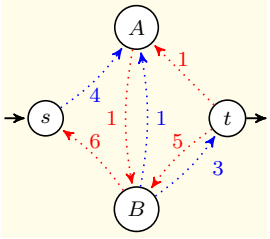
et $A_f = \{(u,v) \in V \times V \mid r_f(u,v) > 0\}$.

Exemple 4

Flot dans le réseau



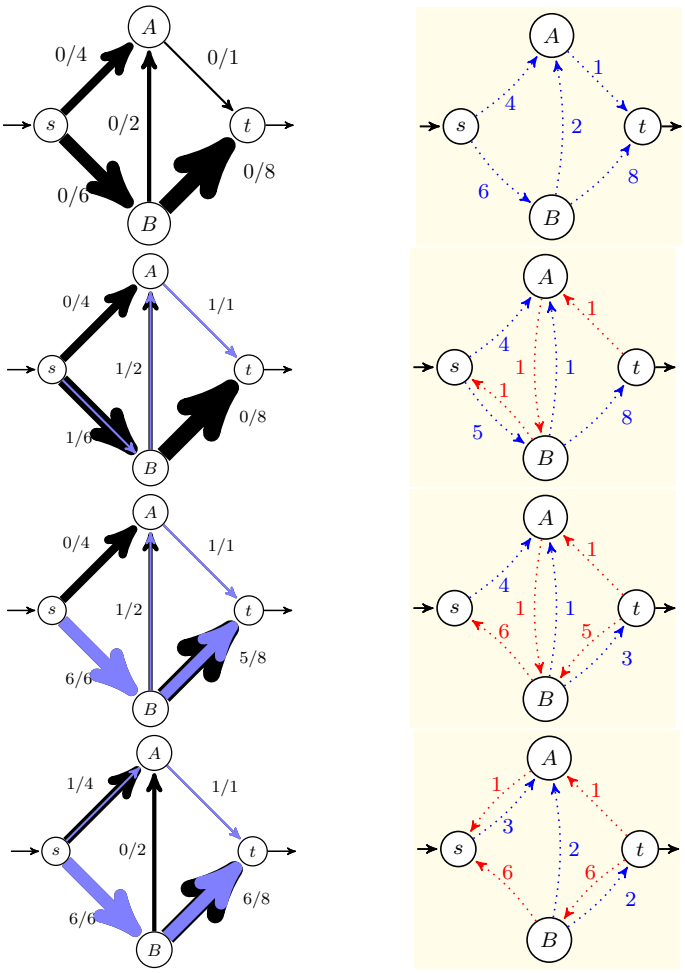
Graphe résiduel correspondant



4 Algorithme de Ford-Fulkerson

pre : Un réseau G
post : Un flot f maximal
fonction fordFulkerson(G)
  f := flot nul
  tant que il existe un chemin simple γ reliant s à t dans le graphe résiduel R_f
    soit γ un tel chemin
    $\Delta = \min\{r_f(u,v) \mid \text{l'arc } u \to v \text{ dans le chemin } \gamma\}$
    pour tout arc $u \to v$ dans γ
      si $u \to v \in E$ **alors**
        $f(u,v) := f(u,v) + \Delta$
      sinon
        $f(v,u) := f(v,u) - \Delta$
  renvoyer f

Exemple 5



Théorème 7 On a l'invariant : « f est un flot ».

Théorème 8 La valeur du flot $|f|$ augmente strictement à chaque itération.

Démonstration.
$f \xrightarrow{\text{itération}} f'$. Soit x le sucesseur de s dans le chemin γ. On a :

$$|f'| = \sum_{v \in V} f'(s,v) = \sum_{v \in V, v \neq x} f(s,v) + f'(s,x) = \sum_{v \in V, v \neq x} f(s,v) + f(s,x) + \Delta = |f| + \Delta.$$

■

5 Capacités entières

Définition 9 (flot à valeurs entières) Un flot f est entier si $f(u,v)$ est un entier pour tout u, v.

Théorème 10 Si les capacités sont des entiers, l'algorithme termine et renvoie un flot entier.

Démonstration. Par l'absurde, supposons que l'algorithme ne termine pas. Les flots calculés par l'algorithme sont tous à valeurs entières. Considérons la suite des valeurs prises par $|f|$.

— Elle est strictement croissante ;
— Elle est majorée par $|f^*|$ où f^* est un flot maximal ;
— Elle est à valeur entière.

C'est une suite infinie d'entiers strictement croissante et majorée. Contradiction. ■

Proposition 11 L'algorithme de Ford-Fulkerson est en $O(|f^*|(V+E))$ où f^* est un flot maximal.

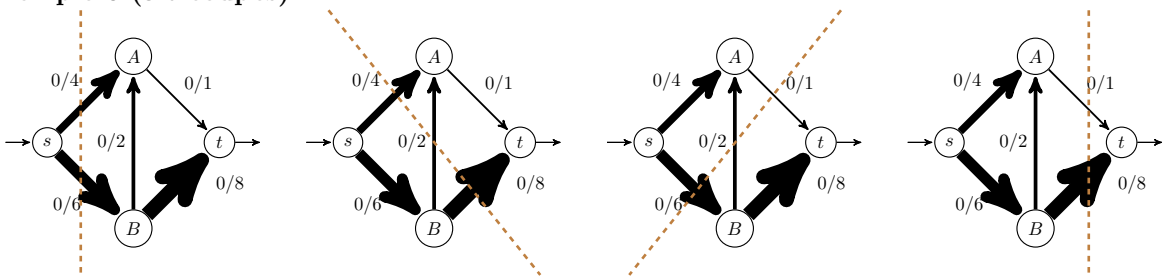
6 Correction de Ford-Fulkerson

Pour démontrer la correction de Ford-Fulkerson, on introduit un certificat d'optimalité : une coupe minimale.

6.1 Coupes

Définition 12 (s-t-coupe) Une s-t-coupe (E,T) d'un réseau est une partition de V avec $s \in E$ et $t \in T$.

Exemple 6 (s-t-coupes)



Définition 13 (capacité d'une coupe) On note $c(E,T) = \sum_{(u,v)\in E\times T} c(u,v)$.

Exemple 7 Les capacité des coupes ci-dessus sont

4+6=10 4+2+8=14 6+1=7 1+8=9

6.2 Flot maximal ≤ Coupe minimale

Définition 14 Soit f un flot. Soit A et B des ensembles de sommets. On note $f(A,B) = \sum_{(u,v)\in A\times B} f(u,v)$.

Lemme 15 Soit f un flot. Soit (E,T) une coupe. On a $|f| = f(E,T) - f(T,E)$.

Démonstration.

$$\begin{aligned}
|f| &= f(s,V) \\
&= f(s,V) - \underbrace{f(V,s)}_{\substack{0 \text{ car } s \text{ pas d'arcs entrants}}} + \sum_{u\in E, u\neq s} \underbrace{f(u,V) - f(V,u)}_{0 \text{ par Kirchhoff}} \\
&= \sum_{u\in E} f(u,V) - f(V,u) \\
&= f(E,V) - f(V,E) \\
&= f(E,T) + f(E,E) - f(E,E) - f(T,E) \\
&= f(E,T) - f(T,E).
\end{aligned}$$

■

Corollaire 16 Pour tout flot f, pour toute coupe (E,T), on a $|f| \leq c(E,T)$.

Démonstration. Le lemme 10 donne $|f| = f(E,T) - f(T,E)$. Donc $|f| \leq f(E,T) \leq c(E,T)$. ■

6.3 Flot maximal ≥ Coupe minimale

Théorème 17 Soit G un réseau et f un flot. On a équivalence entre :

1. Le flot f est maximal
2. Il n'existe pas de chemin de s à t dans le graphe résiduel R_f.
3. Il existe une coupe (E, T) tel que $|f| = c(E, T)$.

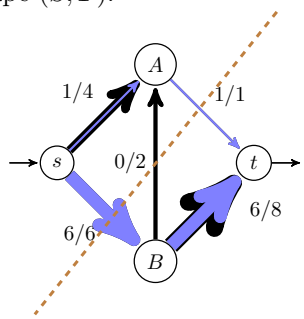
Démonstration.
$\boxed{1 \Rightarrow 2}$ S'il y avait un tel chemin, une itération de Ford-Fulkerson améliorerait le flot.

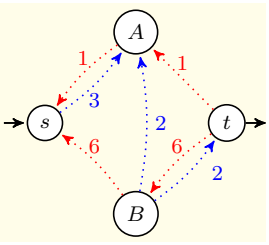
$\boxed{3 \Rightarrow 1}$ Cela signifie que l'on a égalité dans le corollaire 11 et donc f maximal.

$\boxed{2 \Rightarrow 3}$ Supposons qu'il n'y a pas de chemin améliorant. Définissons la coupe (S, T) candidate suivante avec

$$S = \{u \in V, \text{ il existe un chemin de } s \text{ à } u \text{ dans } R_f\}$$

et $T = V \setminus S$. Comme il n'existe pas de chemin améliorant, par définition de S, $t \notin S$ donc $t \in T$. Nous avons bien une coupe (S, T).





Montrons que $|f| = c(S, T)$. Le tableau suivant explique le calcul de $|f|$ en s'appuyant sur la formule $|f| = f(S, T) - f(T, S)$ donnée dans le lemme 10. On considère $u \in S$ et $v \in T$. Par définition de S, on a $u \to v \notin R_f$. Sinon, on aurait un chemin de s à v en passant par u dans R_f et donc $v \in S$. Contradiction.

	$f(u, v)$	$f(v, u)$
u v $u \to v \notin G$ et $v \to u \notin G$	$0 = c(u, v)$	0
$u \to v$ $u \to v \in G$ et $v \to u \notin G$	$c(u, v)$ car capacité $u \to v$ pleine	0
$u \leftarrow v$ $v \to u \in G$ et $u \to v \notin G$	$0 = c(u, v)$ car pas d'arc anti-parallèle	0 car rien à annuler pour $v \to u$

Donc $|f| = f(S, T) - f(T, S) = \sum_{(u,v) \in S \times T} c(u, v) \; - \; 0 = c(S, T)$.

■

Corollaire 18 Ford-Fulkerson est correct.

Démonstration. Quand Ford-Fulkerson s'arrête, il n'existe plus de chemin de s à t dans le graphe résiduel. D'après le théorème précédent, f est donc un flot maximal dans G. ■

Théorème 19 (principe de dualité) Valeur d'un flot maximal = capacité d'une coupe minimale.

Démonstration. $\leq$ par le corollaire 11, et $\geq$ par le théorème 12. ■

7 Algorithme d'Edmonds-Karp (*)

C'est Ford-Fulkerson que l'on spécialise en trouvant un plus court chemin γ en nombre d'arcs dans le graphe résiduel. On peut réaliser cela avec un parcours en largeur.

Théorème 20 L'algorithme d'Edmonds-Karp est en $O(VE^2)$.

Démonstration.

Lemme 21 (monotonie) Pour tout sommet v, la distance de plus court chemin $\delta_f(s,v)$ dans le graphe résiduel R_f croît à chaque itération de l'algorithme.

DÉMONSTRATION. Par l'absurde. Supposons qu'il existe v et un certain moment où le flot passe de f et f' et $\delta_f(s,v) > \delta_{f'}(s,v)$ (*). Prenons un tel sommet v avec $\delta_f(s,v)$ minimal avec (*) (c'est la condition 27.7 dans Cormen).

Soit γ' un plus court chemin de s à v dans $R_{f'}$ et soit u le prédécésseur de v dans ce chemin :

$$s \to \cdots \to u \to v$$

Comme v a été choisi avec $\delta_f(s,v)$ minimal, le sommet u ne satisfait pas la propriété (*), i.e. $\delta_f(s,u) \leq \delta_{f'}(s,u)$ (**).

On a $f(u,v) = c(u,v)$. En effet, par l'absurde, sinon $f(u,v) < c(u,v)$ et il y a un arc (u,v) dans le graphe R_f. Ainsi,

$$\delta_f(s,v) \leq \delta_f(s,u) + 1 \leq_{\text{par (**)}} \delta_{f'}(s,u) + 1 =_{\text{voir } \gamma'} \delta_{f'}(s,v).$$

Contradiction avec (*).

Soit γ le chemin améliorant pour aller de f à f'. Comme (u,v) est dans le graphe résiduel $R_{f'}$, mais pas dans R_f. Donc (v,u) est dans R_f et c'est là qu'il y a eu une modification : (v,u) apparaît dans γ :

$$s \to \cdots \to v \to u \to \ldots$$

Ainsi :

$$\delta_f(s,v) =_{\text{voir } \gamma} \delta_f(s,u) - 1 \leq_{(**)} \delta_{f'}(s,u) - 1 \leq_{\text{voir } \gamma'} \delta_{f'}(s,v) - 2.$$

Contradiction avec (*).

■

Lemme 22 Le nombre d'itérations est $O(VE)$.

DÉMONSTRATION. Un arc (u,v) du graphe résiduel est dit **critique** pour un chemin améliorant γ si la capacité min de γ est atteinte pour (u,v). On montre que tout arc est critique au plus $O(V)$ fois. Comme il y a $O(A)$ arcs, le nombre d'itérations est est en $O(VE)$.

Considérons un moment où (u,v) est critique et que f est le flot à ce moment. Le chemin améliorant est :

$$s \to \cdots \to u \to v \to \ldots$$

Juste après l'arc (u,v) disparaît. Pour qu'il réapparaisse plus tard, il faut que le flot entre u et v diminue si $(u,v) \in A$ ou que le flot augmente entre u et v si $(u,v) \notin A$. Ainsi, quelques moments plus tard, l'arc (v,u) est dans un chemin améliorant et f' ce flot. Ainsi :

Ainsi,

$$\delta_{f'}(s,u) =_{\text{chemin améliorant,}} \delta_{f'}(s,v) + 1 \geq_{\text{monotonie}} \delta_f(s,v) + 1 =_{\text{chemin améliorant}} \delta_f(s,u) + 2$$

$\delta_{f'}(s,u)$ augmente strictement à que fois que (u,v) est critique. Il ne peut que l'être au plus $O(V)$ fois !

■ ■

8 FAQ

Est-ce que le théorème de dualité max flow - min cut est encore vrai pour des capacités réelles ? Oui ! En fait, les capacités entières ne sont utiles que pour la terminaison de l'algorithme. Mais le théorème qui fait le lien entre l'absence de chemin simple de s à t dans le graphe résiduel et le fait qu'on ait un flow max ne demande pas à ce que les capacités soient entières.

Est-ce qu'il existe de meilleurs algorithmes que Ford-Fulkerson (et Edmonds-Karp) ? Oui, voir Wikipedia.

Notes bibliographiques

Ce cours est inspiré de [CLRS09] et de [KT06]. Il y a une littérature énorme sur les flots, même un livre dédié au sujet : [AMO93]. Ford et Fulkerson est à l'origine du problèmes du flot maximum et du couplage biparti [FF62]. De manière indépendante, Edmonds et Karp [EK72], mais aussi Dinic [Din70], ont montré que l'algorithme est polynomial si on utilise le parcours en largeur pour trouver un chemin améliorant dans le graphe résiduel. L'algorithme de Edmonds-Karp est le même algorithme que Ford-Fulkerson mais en choisissant à chaque fois un plus court chemin en nombre d'arcs dans G_f (parcours en largeur). L'algorithme de Edmonds-Karp résout le problème du flot maximum en $O(SA^2)$. Il existe des algorithmes plus performants, basés sur d'autres idées comme les flots bloquants et la méthode pousser-réétiqueter (voir [CLRS09]).

Le calcul d'un flot maximum est utilisé en bioinformatique [PPA+15] pour de la reconstruction de données à partir de brins d'ARN. Il est utilisé pour calculer de chemins sans collision pour un système de plusieurs agents anonymes [YL12]. Il est aussi utilisé en segmentation d'images [KT06].

Références

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[Din70] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[EK72] Jack R. Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2) :248–264, 1972.

[FF62] DR Fulkerson and LR Ford. *Flows in networks*. Princeton University Press, 1962.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[PPA+15] Mihaela Pertea, Geo M Pertea, Corina M Antonescu, Tsung-Cheng Chang, Joshua T Mendell, and Steven L Salzberg. Stringtie enables improved reconstruction of a transcriptome from rna-seq reads. *Nature biotechnology*, 33(3) :290–295, 2015.

[YL12] Jingjin Yu and Steven M. LaValle. Multi-agent path planning and network flow. In Emilio Frazzoli, Tomás Lozano-Pérez, Nicholas Roy, and Daniela Rus, editors, *Algorithmic Foundations of Robotics X - Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics, WAFR 2012, MIT, Cambridge, Massachusetts, USA, June 13-15 2012*, volume 86 of *Springer Tracts in Advanced Robotics*, pages 157–173. Springer, 2012.

Réduction vers le problème du flot maximum

François Schwarzentruber

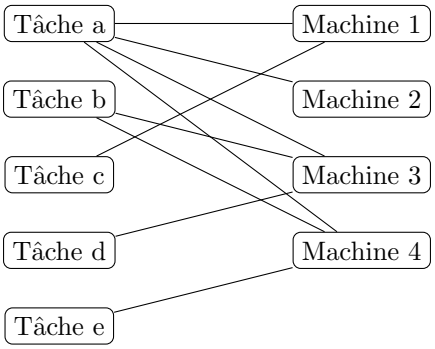
5 octobre 2023

1 Réduction du problème du couplage maximum biparti vers flots

1.1 Couplage maximum dans un graphe biparti

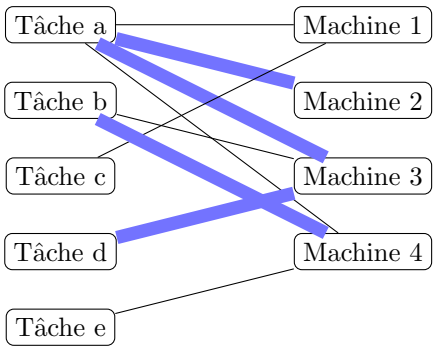
Définition 1 Un graphe non orienté $G = (V, E)$ est biparti s'il existe une partition $V = V_1 \sqcup V_2$ telle que $E \subseteq V_1 \times V_2$.

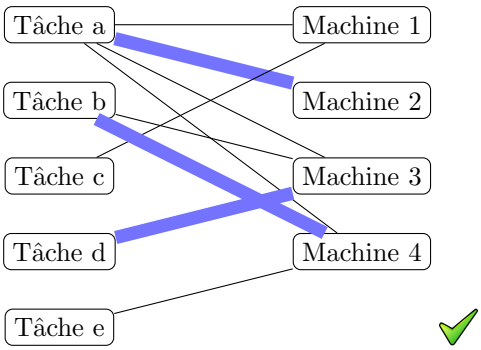
Exemple 1



Définition 2 (couplage) Un couplage dans un graphe $G = (V, E)$ est un ensemble M tel que tout sommet apparaît au plus dans un couple de M.

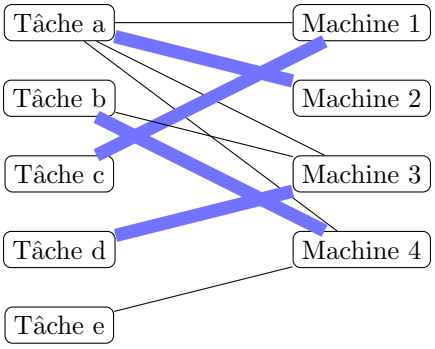
Exemple 2





Définition 3 (couplage maximum) Un couplage M est maximum dans G si $|M|$ est maximal, i.e. pour tout couplage M' dans G on a $|M'| \leq |M|$.

Exemple 3 (couplage maximum) Voici un couplage maximum :



Définition 4 (problème du couplage maximum dans un graphe biparti)
— Entrée : un graphe biparti G ;
— Sortie : un couplage maximum M de G.

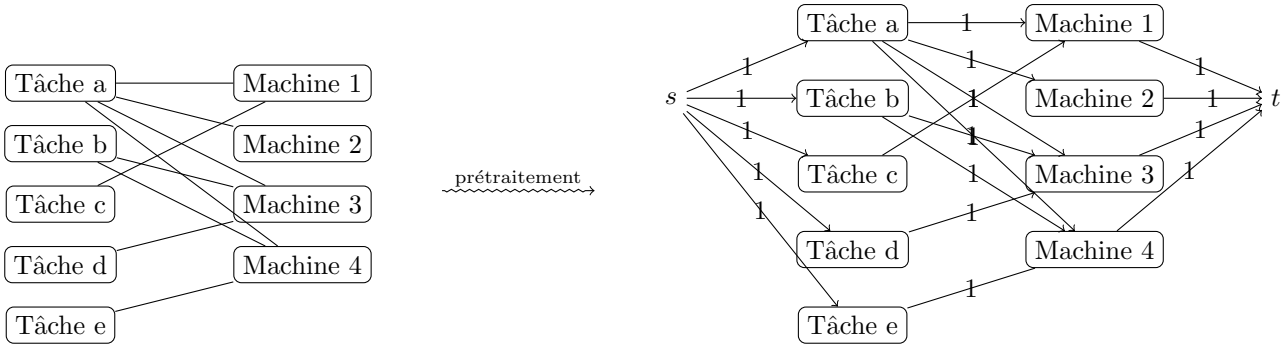
1.2 Réduction aux flots

Nous allons voir comment utiliser les flots pour trouver un couplage maximum dans un graphe biparti. On va faire cela en transformant un graphe biparti en un réseau. De manière plus pompeuse, nous allons transformer une instance du problème du couplage maximum dans un graphe biparti (c'est-à-dire un graphe biparti) en une instance du problème du flot maximal (c'est-à-dire un réseau). On dit que l'on va *réduire* le problème du couplage maximum biparti dans le problème du flot maximal à valeurs entières.

On peut voir ça comme un *pré-traitement*.

1.2.1 Prétraitement

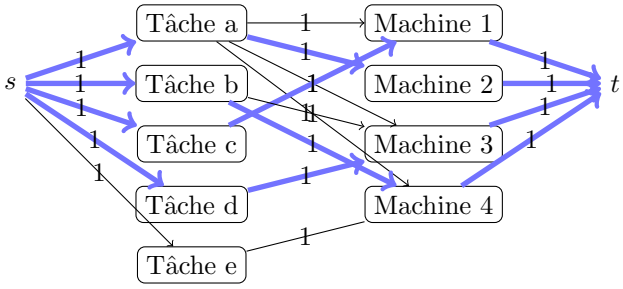
Le prétraitement consiste à ajouter une source s que l'on connecte à tous les sommets de V_1, et à ajouter une destination t à laquelle on connecte tous les sommets de V_2.



Définition 5 (prétraitement dans la réduction) La fonction de prétraitement *pretr* est définie par $pretr((V_1 \sqcup V_2, E)) = (V', E', c, s, t)$ avec :
— $V' = V_1 \sqcup V_2 \sqcup \{s, t\}$;
— $E' = E \cup \{(s, v) \mid v \in V_1\} \cup \{(u, t) \mid u \in V_2\}$;
— $c(u, v) = 1$ pour tout $(u, v) \in E'$.

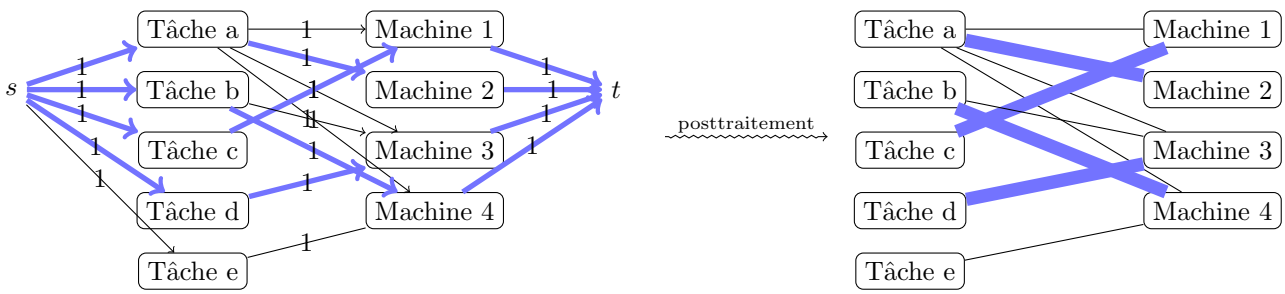
1.2.2 Résolution

On résout par exemple avec Ford-Fulkerson.



1.2.3 Post-traitement

Maintenant, on veut le couplage. Le post-traitement consiste à sélectionner les arêtes où l'eau passe.



Définition 6 (posttraitement dans la réduction) La fonction de prétraitement *posttr* est définie par

$$posttr(f) = \{(u, v) \in V_1 \times V_2 \mid f(u, v) = 1\}.$$

1.3 Correction

Proposition 7 Étant donné un flot f maximal à valeurs entières dans $pretr(G)$. L'ensemble d'arêtes $posttr(f)$ est un couplage maximum dans G.

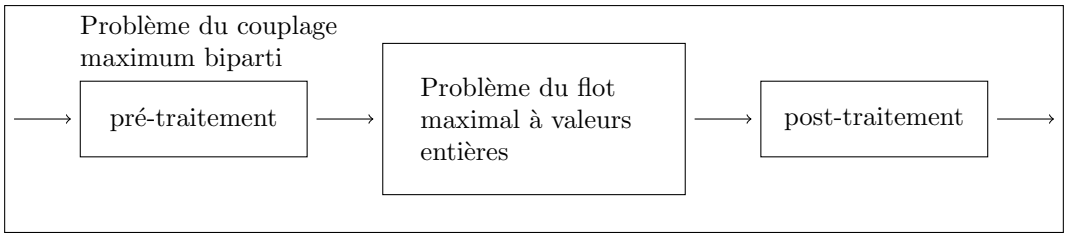
Démonstration. On montre que |couplage maximal| = |flot max|.

$\boxed{\leq}$ Soit M un couplage maximal. J'en déduis un flot en faisant passer de l'eau par les arêtes de M. Donc le couplage maximal est minoré par le flot maximal.

$\boxed{\geq}$ Sit f est le flot maximal, d'après l'exécution de l'algorithme de Ford-Fulkerson, il existe une solution entière. Les arêtes centrales sont soit à 0 ou 1. En définissant M comme l'ensemble des arêtes centrale où le flot vaut 1, on obtient un couplage. Donc le flot maximal minore le couplage maximal.

L'algorithme renvoie bien un couplage et sa valeur est |flot max|. ■

1.4 Bilan

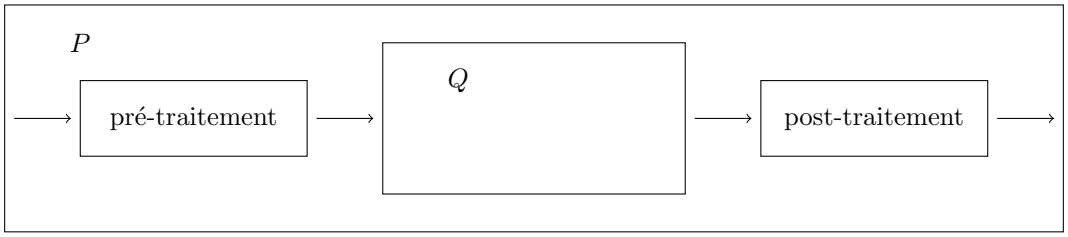


Définition 8 (réduction avec pré- et post-traitements) Soit P, Q deux problèmes algorithmiques. Une réduction de P à Q est la donnée :

— d'un algorithme de pré-traitement qui transforme une instance de P en une instance Q ;

— d'un algorithme de post-traitement qui transforme une solution de Q en une solution de P

telle que la boîte suivante donne un algorithme correct pour P :

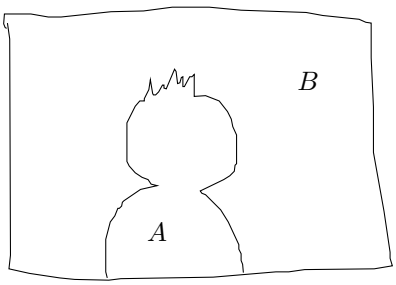


Remarque 9 Dans un graphe général (pas forcément biparti), le problème du couplage maximal se résout en temps polynomial avec l'algorithme d'Edmonds.

2 Segmentation d'images

Fait en cours. Implémentation en TD

La segmentation d'images consiste à prendre une image bitmap (une photo par exemple) et à séparer l'avant-plan de l'arrière plan.



2.1 Définition

— entrée :
  — une image bitmap, où chaque pixel numéro i est tagué avec une vraisemblance a_i d'être dans l'avant-plan, et une vraisemblance b_i d'être dans l'arrière-plan ;
  — pour toute paire de pixels voisins (i, j), on a une mesure de similarité de $sim_{ij} \geq 0$ de pas mettre i et j tout deux dans l'avant-plan, ou tout deux dans l'arrière-plan.
— sortie : une partition des pixels (A, B) qui maximise

$$q(A, B) = \sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{(i,j) \in E \cap (A \times B \cup B \times A)} sim_{ij}.$$

En gros, on additionne les vraissemblances pour chaque pixel et on pénalise avec sim_{ij} si les deux pixels i et j voisins ne sont pas classés de la même façon (tout deux dans l'avant-plan ou tout deux dans l'arrière-plan).

2.2 Prétraitement

D'abord, nous allons réécrire $q(A, B)$ à maximiser en isolant une quantité à minimiser (afin de se rapprocher d'une coupe minimale).

$$q(A, B) = \sum_i a_i + b_i - \left(\underbrace{\sum_{i \in A} b_i + \sum_{i \in B} a_i + \sum_{(i,j) \in E \cap (A \times B \cup B \times A)} sim_{ij}}_{q'(A,B)} \right)$$

Maximiser $q(A, B)$ revient à minimiser $q'(A, B)$.
On construit alors le réseau suivant :
— les sommets sont les pixels i + une source s et une destination t
— Les arcs sont :
  — (s, i) de capacité a_i ;
  — (j, t) de capacité b_j ;
  — des arcs entre pixels voisins i, j : (i, j) et (j, i) tout deux de capacité sim_{ij}.

2.3 Posttraitement

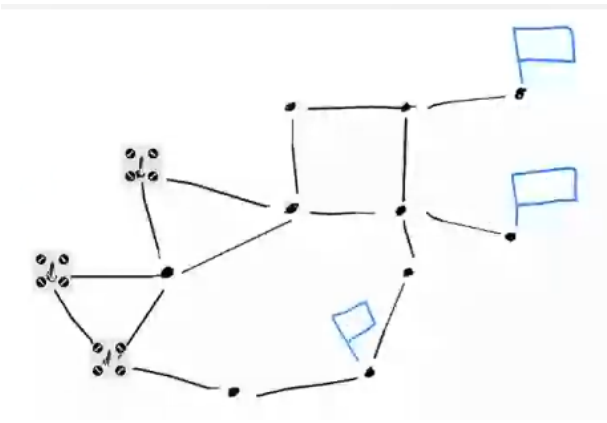
En calculant une coupe minimale, on récupère la segmentation. De manière générale, une coupe est de la forme

$$c(A, B) = (\{s\} \cup A, B \cup \{t\})$$

où A, B est une partition de l'ensemble des pixels. Sa capacité est $q'(A, B)$.
Ainsi, calculer une coupe minimale revient à miniser $q'(A, B)$.

3 Déplacement de robots anonymes

Un champ d'investigation est l'utilisation de plusieurs robots dans des entrepots. Plusieurs problèmes sont appelé sous la dénomination MAPF pour *multi-agent path finding*. Les n robots ont des positions de départs $(s_1, \ldots, s_n)$ et doivent attendre des destinations $(t_1, \ldots, t_n)$. Généralement, l'environnement est modélisé par un graphe $G = (V, E)$ non orienté.



On cherche à calculer des chemins $\pi_i = (\pi_{i,0}, \ldots, \pi_{i,k})$ pour chaque robot i :
— $\pi_{i_0} = s_i$
— pour tout temps $\ell \in \{0, \ldots, k-1\}$, $\pi_{i,\ell} = \pi_{i,\ell+1}$ ou $(\pi_{i,\ell}, \pi_{i,\ell+1}) \in E$.

De plus, on demande à ce que les agents atteignent une cible chacun, mais de façon anonyme. Il existe une permutation σ telle que pour tout i, $\pi_{i,k} = t_{\sigma i}$.

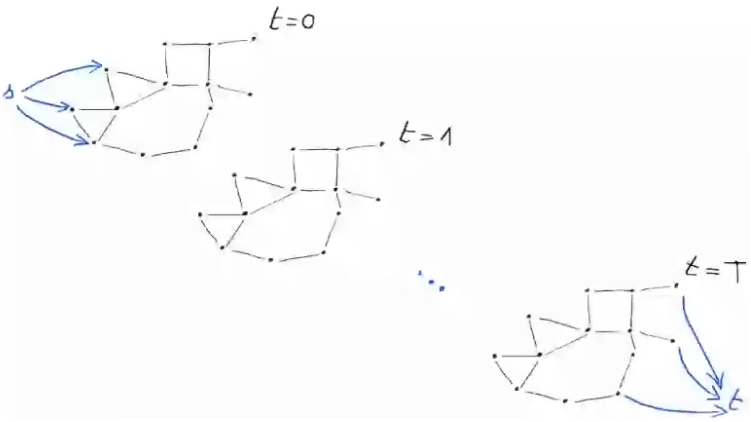
Aussi, les robots n'entrent pas en collision. Pour tout temps ℓ, pour tout i, j, $\pi_{i,\ell} \neq \pi_{j,\ell}$.

De tels chemins s'appellent un plan de déplacement à k étapes pour $(G, s_1, \ldots, s_n, t_1, \ldots, t_n)$.

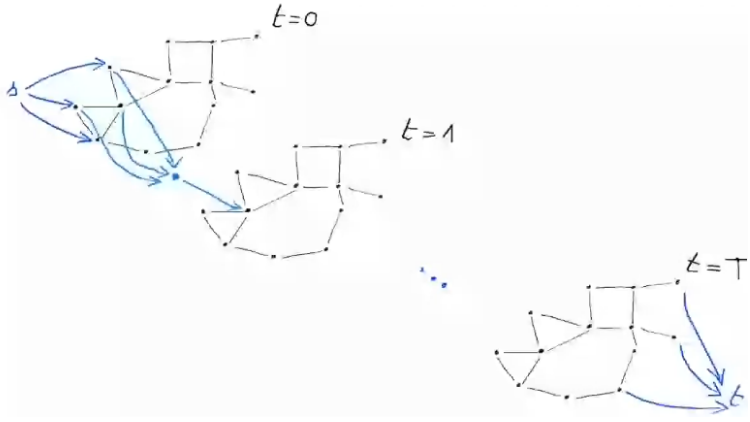
Définition 10 (MAPF anonyme) Une instance est un tuple où G est
— entrée : une instance $(G, s_1, \ldots, s_n, t_1, \ldots, t_n)$;
— sortie : un plan de déplacement de k étapes pour $(G, s_1, \ldots, s_n, t_1, \ldots, t_n)$.

3.1 Réduction

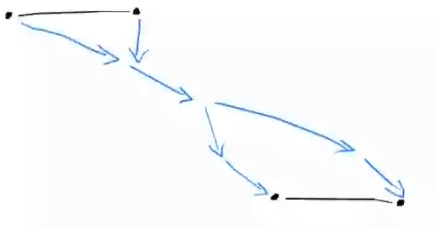
On crée un réseau comme suit.
— on crée une super-source, on crée une super-destination
— on crée k copies de G
— on relie la super-source au sommet $s_1, \ldots, s_n$ dans la première copie de G
— on relie les sommets $t_1, \ldots, t_n$ de la dernière copie de G vers la super-destination.



Maintenant, pour éviter les collisions, pour les arcs de G, on construit un gadget entre deux copies successives de G pour assurer qu'il n'y a pas de collisions :



On peut aussi encoder qu'il n'y a pas de switch de deux robots sur une même arête :

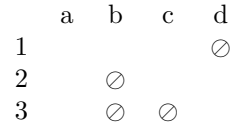


4 Placement de robots et théorème de König(*)

fait en TD

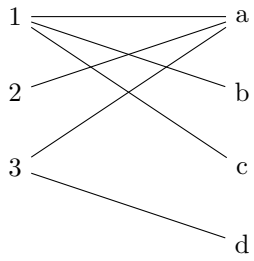
4.1 Placements de tours vers couplage

Entrée : une grille avec des cases interdites :

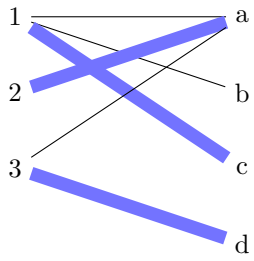


Sortie : placement d'un nombre maximal de tours (du jeu d'échecs) sur les cases autorisées sans qu'elles puissent s'attaquer.

On modélise le problème comme un problème de couplage maximal. On considère le graphe biparti : lignes VS colonnes. Les arcs sont les cases autorisées de la grille.



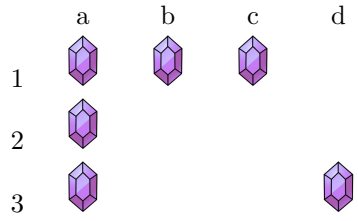
Une solution est par exemple :



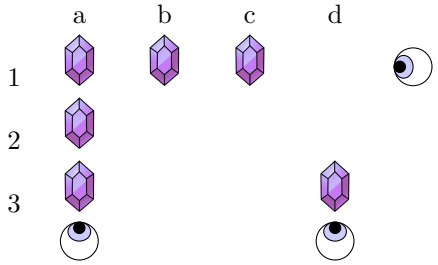
Sur la grille ça donne :

4.2 Robots vers minimum vertex cover vers couplage

Cette section provient d'un exercice proposé au concours de programmation SWERC 2012. Chaque robot peut contrôler une ligne ou une colonne. On souhaite placer un nombre minimal de robots pour que tous les trésors  soient contrôlés par au moins un robot. Voici une entrée possible du problème :



On peut alors placer des robots comme ceci :



Le problème revient à savoir dans quelles lignes et quelles colonnes placer le moins de robots de façon à couvrir tous les emplacements. Ce problème se réduit au problème du *minimum vertex cover* (couverture de sommets minimal) défini de la façon suivante :

— Entrée : un graphe non orienté ;

— Sortie : oui ssi un ensemble minimal de sommets qui touchent toutes les arrêtes.

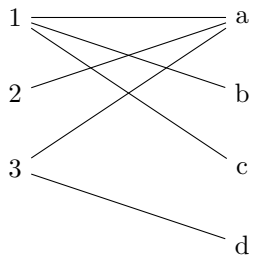
On construit le graphe suivant :

— Les sommets sont les lignes et les colonnes.

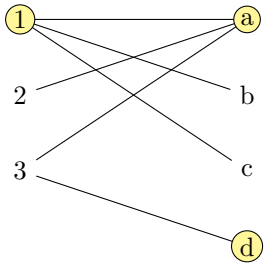
— Les arcs sont les rubis, c'est à dire on met un arc entre un sommet "ligne" et un sommet "colonne" s'il y a un rubis à l'intersection.

Le problème revient alors à trouver un ensemble minimal de sommets (les "lignes" ou "colonnes" où placer les robots) qui touchent toutes les arrêtes (i.e. voir toutes les étoiles).

L'exemple donne le graphe suivant :



Une couverture d'ensemble minimale est par exemple :



Théorème 11 (de König) Dans un graphe biparti, la cardinalité minimale d'une couverture de sommets est égale au couplage maximal.

Démonstration. $\boxed{\geq}$ Cette inégalité est valable dans tout graphe G, non nécessairement biparti. Soit C une couverture. Soit M un couplage. On a $|M| \leq |C|$. En effet, si M est un couplage (i.e. un ensemble d'arêtes disjointes), et C une couverture (i.e. un ensemble de sommets qui touchent toutes les arêtes), on a

$$M = \bigcup_{u \in C} A_u$$

où A_u est vide ou alors égal à l'unique arête de M touché par u. Donc

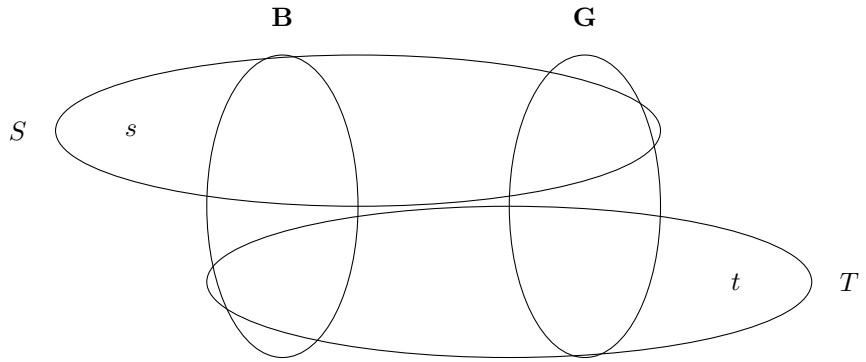
$$|M| \leq \sum_{u \in C} |A_u| \leq \sum_{u \in C} 1 = |C|.$$

En passant au max sur M et min sur C, on a que le couplage maximal est inférieur ou égal à la cardinalité minimale d'une couverture de sommets.

$\boxed{\leq}$ On utilise ici le fait que G est biparti. Pour calculer le couplage maximal, on construit le réseau de flot, mais ici on va mettre des arcs de capacité infinie entre les "garçons" et les "filles". Les capacités entre la source et chaque garçon reste à 1 et les capacités entre les filles et la destination reste à 1 également.

Le couplage maximal est égal au flot maximal, qui est majoré par le nombre de garçons, donc fini. Ainsi, tous les résultats démontrés précédemment dans ce chapitre restent valables, dont le théorème du flot maximal et coupe minimale. Le flot maximal est égal à la coupe minimale.

Soit (S, T) une coupe minimale, i.e. une coupe telle que $c(S, T) = |f^*|$ où f^* est un flot maximal. Soit $\mathbf{B}$ l'ensemble des sommets garçons et $\mathbf{G}$ l'ensemble des sommets filles. Montrons que $C = (\mathbf{B} \cap T) \cup (\mathbf{G} \cap S)$ est un ensemble de sommets couvrants.



Lemme 12 C touche toutes les arêtes, i.e. c'est une couverture.

DÉMONSTRATION. Cela revient à montrer qu'il n'existe pas d'arêtes de $\mathbf{B} \cap S$ vers $\mathbf{G} \cap T$. Cela n'est pas le cas, car une telle arête, de capacité infinie, serait comptabilisée dans $c(S,T)$ et donc $c(S,T) = +\infty$. Contradiction, car $c(S,T)$ est fini (car égal au flot maximal par le théorème du flot maximal / coupe minimal). ■

Lemme 13 $c(S,T) = |C|$.

DÉMONSTRATION. $c(S,T)$ est, par définition, la somme du nombre d'arcs qui vont de s dans $\mathbf{B} \cap T$ et ceux qui vont de $\mathbf{G} \cap S$ à t. Cette somme vaut le cardinal de C. On a donc $c(S,T) = |C|$. ■

Donc $|couverturemin| \leq |(|C) = c(S,T) = |\text{couplage max}|$. ■

5 Élimination dans un tournoi de baseball

5.1 Description informelle du problème

On suppose qu'une équipe gagne un point à chaque fois qu'elle gagne un match (il n'y a pas de matchs nuls). L'entrée du problème est une table qui résume la situation entre les équipes. Dans cette table, on indique le nombre de matchs déjà gagnés pour chaque équipe (parmi les matchs déjà joués). On donne aussi les nombres de matchs qu'il reste à jouer (on donne le nombre de matchs entre chaque équipe). Par exemple :

Equipes	Nombre de matchs déjà gagnés	Nombres de matchs restants			
		logiciens	musiciens	poètes	informaticiens
logiciens	90	-	1	6	4
musiciens	88	1	-	1	4
poètes	87	6	1	-	4
informaticiens	79	4	4	4	-

Là, dans l'exemple, l'équipe des informaticiens a déjà gagné 79 matchs. Il y a encore 6 matchs à organiser entre les poètes et les logiciens.

L'objectif est de connaître les équipes éliminées d'office. Une équipe i est éliminée si, quoiqu'il arrive dans le futur, une autre équipe qui aura strictement plus de points que i. Par exemple, est-ce que l'équipe des informaticiens est éliminée ?

Fait. Dans l'exemple, les informaticiens sont éliminés d'office.

Expliquons pourquoi ils sont éliminés d'office. D'une part, dans le meilleur des cas, les informaticiens gagnent les 4+4+4 = 12 matchs qu'ils leur restent à jouer. Ainsi, ils ramassent un gain maximal de $79 + 12 = 91$.

D'autre part, on montre que soit l'équipe des logiciens soit l'équipe des poètes gagne strictement plus de 91. Pour le voir, on considère le nombre de points gagnés à eux deux, i.e. comme si c'était une méta-équipe. En effet, à eux deux, ils ont déjà $90 + 87 = 177$ points. À la fin, les logiciens et les poètes n'auront plus que jouer l'un contre l'autre. Il y a encore 6 matchs les opposant, donc ils gagneront à eux deux 6 points supplémentaires. En résumé, à la fin, à eux deux, ils auront $177 + 6 = 183$ points. L'une des deux équipes (logiciens ou poètes) aura plus de $\frac{183}{2} > 91$ points.

5.2 Description formelle du problème

Equipes	Nombre de matchs déjà gagnés	Nombres de matchs restants			
		1	2	3	4
1	90	-	1	6	4
2	88	1	-	1	4
3	87	6	1	-	4
4	79	4	4	4	-

Soit $\mathcal{E}$ un ensemble fini d'équipes. On note p_i le nombre de points déjà gagnés de l'équipe i. Pour toutes équipes i et $j \neq i$, on note $r_{\{i,j\}}$ le nombre de matchs qu'il reste à jouer entre i et j. On remarque que l'indice $\{i,j\}$ dans $r_{\{i,j\}}$ est un sous-ensemble de taille 2, et donc $r_{\{i,j\}} = r_{\{j,i\}}$.

Dans l'exemple, $\mathcal{E} = \{1,2,3,4\}$, $p_2 = 88$ et $r_{\{1,3\}} = 6$.

Pour tout sous-ensemble d'équipes $T \subseteq \mathcal{E}$, on note $M_T = \{\{j,k\} \mid j,k \in T \text{ and } j \neq k\}$, qui est l'ensemble des configurations de matchs possibles entre les équipes dans T. Une équipe i est éliminée si on arrive à montrer qu'une autre équipe parmi un sous-ensemble d'équipes T aura strictement plus de points que i à la fin du tournoi. On note $maxp_i$ le nombre maximal de points que i peut espérer avoir en fin de tournoi :

$$maxp_i = p_i + \sum_{k \in \mathcal{E}, k \neq i} r_{\{i,k\}}.$$

Formellement :

Définition 14 Une équipe $i \in \mathcal{E}$ est *éliminée* s'il existe un sous-ensemble $T \subseteq \mathcal{E}$ non vide tel que

$$\frac{\sum_{j \in T} p_j + \sum_{m \in M_T} r_m}{|T|} > maxp_i.$$

Le numérateur $\sum_{j \in T} p_j + \sum_{m \in M_T}$ est le gain gagné par T considéré comme une méta-équipe. En divisant par $|T|$, on obtient la moyenne des gains des équipes de T. On sait donc qu'une des équipes dans T a au moins plus que $\frac{\sum_{j \in T} p_j + \sum_{m \in M_T} r_m}{|T|}$. L'inégalité dit alors que i est bel et bien éliminée par cette équipe.

Exemple 15 Dans l'exemple, l'équipe 4 est éliminée car pour $T = \{1,3\}$, on a

$$\frac{90 + 87 + 6}{2} = 91.5 > 79 + 4 + 4 + 4 = 91.$$

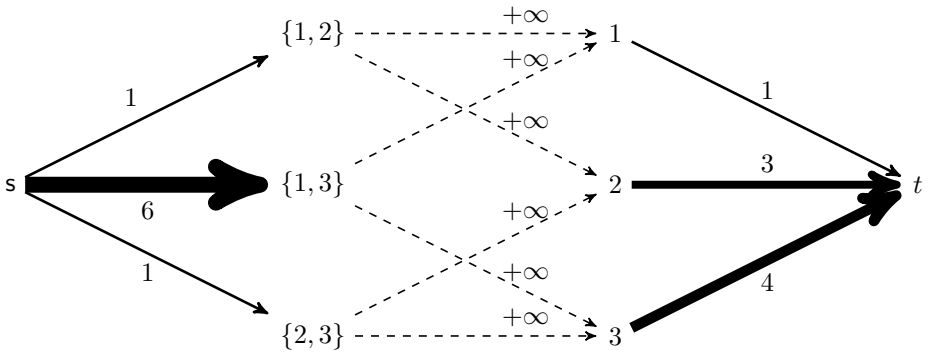
Définition 16 Le problème algorithmique de l'élimination dans un tournoi de baseball est :
- entrée : un ensemble fini d'équipes $\mathcal{E}$; pour toute équipe i, p_i, le nombre de points déjà gagnés par i ; pour toute paire d'équipes différentes $i \neq j$, $r_{\{i,j\}}$ le nombre de matchs entre i et j ; une équipe i particulière ;
- sortie : oui si i est éliminée, non sinon.

5.3 Prétraitement

Le problème de savoir si une équipe i est éliminée semble difficile car on a l'impression qu'il faut deviner T. Pourtant on peut se ramener au calcul d'un flot. On définit un réseau de flots G_i où les sommets sont : une source s, les configurations de match n'impliquant pas i, les équipes sauf i et une destination t. On relie la source s et une configuration de matchs m avec le nombre restant de matchs entre les équipes de m à jouer comme capacité. On relie chaque configuration de match $\{j,k\}$ aux équipes j et k avec une capacité infinie. On relie chaque équipe j à la destination t avec comme capacité le nombre de points manquants de j pour atteindre le score $maxp_i$. Formellement :

Définition 17 Soit $i \in \mathcal{E}$. On définit un réseau de flots $G_i = (V, E, c, \mathsf{s}, t)$ où
- $V = \{\mathsf{s}, t\} \sqcup (\mathcal{E} \setminus \{i\}) \sqcup M_{\mathcal{E}\setminus\{i\}}$ où $\sqcup$ est le symbole pour 'union disjointe' ;
- $E = \{(\mathsf{s}, m) \mid m \in M_{\mathcal{E}\setminus\{i\}}\} \sqcup \{(m,j) \mid m \in M_{\mathcal{E}\setminus\{i\}} \text{ et } j \in m\} \sqcup \{(j,t) \mid j \in \mathcal{E} \setminus \{i\}\}$;
- $c : E \to \mathbb{R}^+ \sqcup \{+\infty\}$ définie par :
  1. $c(\mathsf{s}, m) = r_m$ pour tout $m \in M_{\mathcal{E}\setminus\{i\}}$;
  2. $c(m,j) = +\infty$ pour tout $m \in M_{\mathcal{E}\setminus\{i\}}$ et $j \in m$;
  3. $c(j,t) = maxp_i - p_j$ pour tout $j \in \mathcal{E} \setminus \{i\}$.

Exemple 18 Sur l'exemple, le réseau de flots G_4 est :



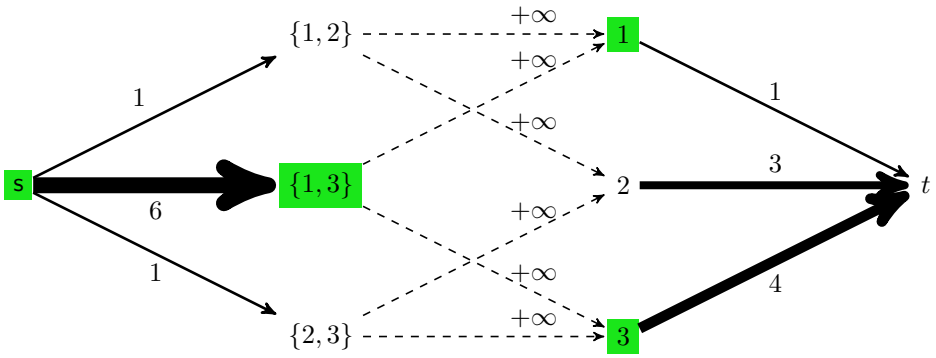
Intuitivement, c'est comme si on faisait toujours gagner l'équipe 4 (car les matchs qui incluent l'équipe 4 ne sont pas mentionnés car ils ne donnent pas de points à d'autres équipes). Le réseau de flots représente les contraintes de distribution des points entre les équipes restantes. La source distribue des points aux configurations de matchs (autant que de matchs qu'il reste à jouer). Puis, une configuration de matchs, par exemple 1-2, distribue les points entre les équipes concernées, à savoir 1 ou 2. Puis, à la fin, le réseau limite l'obtention de points par les équipes pour qu'aucune ne dépasse l'équipe 4.

5.4 Posttraitement

Un flot correspond à une distribution des points sous ces contraintes. En particulier, on demande à ce qu'aucune équipe ne dépasse l'équipe i. En d'autres termes, on demande à ce que l'équipe i ne *soit pas éliminée*. Les points distribuées correspondent aux matchs réalisés entre les équipes (car on suppose que l'équipe i contre l'une d'elles gagne). Ainsi, les points distribuées en tout est $\sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$.

- Si le flot est maximal est strictement inférieure à $\sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$, cela signifie qu'il est impossible de faire jouer tous les matchs sous les contraintes. Les contraintes sont 'insatisfaisables'. Donc l'équipe i est éliminée.
- Si le flot maximal est égal à $\sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$, cela signifie que l'on peut faire jouer tous les matchs entre les équipes sous l'hypothèse que l'équipe i gagne tous ses matchs et sans qu'elles dépassent le score final de i. Donc l'équipe i n'est pas (ou pas encore ;)) éliminée.

Exemple 19 Dans l'exemple ci-dessus, $\sum_{m \in M_{\mathcal{E}\setminus\{4\}}} r_m = 6 + 1 + 1 = 8$. Le flot maximal vaut $7 < 8$. Donc l'équipe 4 est éliminée. Voici une coupe minimale possible :



La coupe vaut $1 + 1 + 1 + 4 = 7$.

5.5 Correction

Théorème 20 *i est éliminée ssi $|f| < \sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$ où $|f|$ est la valeur d'un flot maximal f de G_i.*

Démonstration.
Caractérisons l'élimination d'une équipe en terme de coupe. On note $(S_T, V \setminus S_T)$ la coupe où $S_T = \{\mathsf{s}\} \cup M_T \cup T$. L'exemple 19 montre la coupe $(E_{\{1,3\}}, V \setminus E_{\{1,3\}})$. On a :

Lemme 21 *L'équipe i est éliminée ssi il existe $T \subseteq \mathcal{E}$ tel que $c(S_T, V \setminus S_T) < \sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$.*

Démonstration. Pour tout $T \subseteq \mathcal{E}$, soit $\Delta_T = -\sum_{j \in T} p_j - \sum_{m \in M_T} r_m + |T| \times maxp_i$. On a :

$$\begin{aligned} c(S_T, V \setminus S_T) &= \sum_{m \in M_{\mathcal{E}\setminus\{i\}\setminus S_T}} r_m + \sum_{j \in T} (maxp_i - p_j) \\ &= \sum_{j \notin T \text{ ou } k \notin T} r_{\{j,k\}} + |T| \times maxp_i - \sum_{j \in T} p_j \\ &= \sum_{m | m \in M_{\mathcal{E}\setminus\{i\}}} r_m - \sum_{m \in M_T} r_m + |T| maxp_i - \sum_{j \in T} p_j \\ &= \sum_{m | m \in M_{\mathcal{E}\setminus\{i\}}} r_m + \Delta_T. \end{aligned}$$

Ainsi,

i est éliminée ssi il existe $T \subseteq \mathcal{E}$, $\Delta_T < 0$ (définition 14)
ssi il existe $T \subseteq \mathcal{E}$, $c(S_T, V \setminus S_T) < \sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$ (calcul précédent). ■

$\boxed{\Rightarrow}$ Supposons que $|f| \geq \sum_{m \in \mathcal{E}\setminus\{i\}} r_m$. Montrons que l'équipe i n'est pas éliminée. Pour tout $T \subseteq \mathcal{E}$, $|f| \leq c(S_T, V \setminus S_T)$ car le flot maximal minore toutes les coupes. Donc en utilisant l'hypothèse, pour tout $T \subseteq \mathcal{E}$,

$$c(S_T, V \setminus S_T) \geq \sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m.$$

Par le lemme 21, l'équipe i n'est pas éliminée.

$\boxed{\Leftarrow}$ Supposons que $|f| < \sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$. Montrons que l'équipe i est éliminée. Soit $(S, V \setminus S)$ une coupe minimale. On va construire une coupe minimale de la forme $(S_T, V \setminus S_T)$. Soit $T = S \cap \mathcal{E}$, c'est à dire l'ensemble des sommets de type 'équipe' de S.

Lemme 22 $\{j,k\} \in S$ implique $j \in T$ et $k \in T$.

Démonstration. Par l'absurde, supposons qu'il existe $j, k \in \mathcal{E} \setminus \{i\}$ tel que $\{i,j\} \in S$ et $j \notin T$. Mais l'arc $\{j,k\} - j$ est de capacité $+\infty$. Donc la valeur de la coupe est infinie. Comme la coupe $(S, V \setminus S)$ est minimale, sa valeur est égale au flot et est donc fini. Contradiction. ■

Lemme 23 $(S_T, V \setminus S_T)$ est aussi une coupe minimale.

Démonstration.

$$\begin{aligned} c(S_T, V \setminus S_T) &= \sum_{\{j,k\} | j \notin T \text{ ou } k \notin T} r_{\{j,k\}} + |T| \times maxp_i - \sum_{j \in T} p_j && \text{(déf. de } c(S_T, V \setminus S_T)) \\ &\leq \sum_{\{j,k\} \notin E} r_{\{j,k\}} + |T| \times maxp_i - \sum_{j \in T} p_j && \text{(contraposée du lemme 22)} \\ &= c(S, V \setminus S) && \text{(déf. de } c(S_T, V \setminus S_T)). \end{aligned}$$

■

D'après le théorème coupe minimal/flot maximal, on a $c(S_T, V \setminus S_T) = |f|$. D'après l'hypothèse, on a $c(S_T, V \setminus S_T) < \sum_{m \in M_{\mathcal{E}\setminus\{i\}}} r_m$. D'après le lemme 21, l'équipe i est éliminée. ■

6 FAQ

Est-ce que le théorème de dualité max flow - min cut est encore vrai pour des capacités réelles ? Oui ! En fait, les capacités entières ne sont utiles que pour la terminaison de l'algorithme. Mais le théorème qui fait le lien entre l'absence de chemin simple de s à t dans le graphe résiduel et le fait qu'on ait un flow max ne demande pas à ce que les capacités soient entières.

Est-ce que la réduction pour la segmentation d'images est utilisé ? Elle l'était avant l'utilisation du deep learning. La référence c'est [BVZ01].

Est-ce qu'il existe de meilleurs algorithmes que Ford-Fulkerson (et Edmonds-Karp) ? Bien sûr ! Voir wikipedia.

Est-ce qu'il existe de meilleurs algorithmes pour le problème de couplage dans un graphe biparti ?

Quel algo pour couplage dans un graphe quelconque ?

Et si on met des poids et qu'on veut un couplage de coût min ?

Notes bibliographiques

Ce cours est inspiré de [CLRS09] et de [KT06]. Il y a une littérature énorme sur les flots, même un livre dédié au sujet : [AMO93]. Ford et Fulkerson est à l'origine du problèmes du flot maximum et du couplage biparti [FF62]. De manière indépendante, Edmonds et Karp [EK72], mais aussi Dinic [Din70], ont montré que l'algorithme est polynomial si on utilise le parcours en largeur pour trouver un chemin améliorant dans le graphe résiduel. L'algorithme de Edmonds-Karp est le même algorithme que Ford-Fulkerson mais en choisissant à chaque fois un plus court chemin en nombre d'arcs dans G_f (parcours en largeur). L'algorithme de Edmonds-Karp résout le problème du flot maximum en $O(SA^2)$. Il existe des algorithmes plus performants, basés sur d'autres idées comme les flots bloquants et la méthode pousser-réétiqueter (voir [CLRS09]).

Le calcul d'un flot maximum est utilisé en bioinformatique [PPA+15] pour de la reconstruction de données à partir de brins d'ARN. Il est utilisé pour calculer de chemins sans collision pour un système de plusieurs agents anonymes [YL12]. Il est aussi utilisé en segmentation d'images [KT06].

Références

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

[BVZ01] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11) :1222–1239, 2001.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[Din70] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[EK72] Jack R. Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2) :248–264, 1972.

[FF62] DR Fulkerson and LR Ford. *Flows in networks*. Princeton University Press, 1962.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[PPA+15] Mihaela Pertea, Geo M Pertea, Corina M Antonescu, Tsung-Cheng Chang, Joshua T Mendell, and Steven L Salzberg. Stringtie enables improved reconstruction of a transcriptome from rna-seq reads. *Nature biotechnology*, 33(3) :290–295, 2015.

[YL12] Jingjin Yu and Steven M. LaValle. Multi-agent path planning and network flow. In Emilio Frazzoli, Tomás Lozano-Pérez, Nicholas Roy, and Daniela Rus, editors, *Algorithmic Foundations of Robotics X - Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics, WAFR 2012, MIT, Cambridge, Massachusetts, USA, June 13-15 2012*, volume 86 of *Springer Tracts in Advanced Robotics*, pages 157–173. Springer, 2012.

Programmation linéaire réelle

François Schwarzentruber

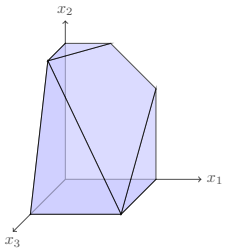
https://playwithalgos.github.io/simplex/

1 Programme linéaire

Exemple 1

Il vend des chocolats simples (1 euro), des pyramides (6 euros) et des pyramides de luxe (13 euros).
Au maximum, il peut vendre 200 chocolats simples, 300 pyramides, pas plus de 400 chocolats en tout et le nombre de pyramides plus trois fois le nombre de pyramides de luxe est au plus 600.

$$\text{maximiser } x_1 + 6x_2 + 13x_3$$
$$\begin{cases} x_1 \leqslant 200 \\ x_2 \leqslant 300 \\ x_1 + x_2 + x_3 \leqslant 400 \\ x_2 + 3x_3 \leqslant 600 \\ x_1, x_2, x_3 \geq 0 \\ x_1, x_2, x_3 \in \mathbb{R} \end{cases}$$



Définition 2 (programme linéaire) Un programme linéaire réel est la donnée :
- d'une fonction objectif linéaire à maximiser ou minimiser ;
- sous des contraintes linéaires (in.égalités), où les variables prennent leurs valeurs dans $\mathbb{R}$.

$$\text{maximiser fonction objectif linéaire}$$
$$\left\{ \text{contraintes linéaires} \right.$$

Définition 3 Une solution est la valeur de variables $x_1, \ldots, x_n$ qui respectent les contraintes.

Définition 4 Une solution optimale est la valeur de variables $x_1, \ldots, x_n$ qui respectent les contraintes qui optimisent la fonction objectif.

Définition 5 **Programmation linéaire réelle**

entrée : un programme linéaire réel P
sortie : une solution optimale de P, non borné s'il n'y a pas d'optimum, ou impossible si les contraintes sont inconsistantes.

Définition 6 Un ensemble S est convexe si pour tout $x, y \in S$, $\lambda x + (1-\lambda)y \in S$.

Proposition 7 Les contraintes forment un ensemble convexe.

Démonstration. Car c'est une intersection de convexe. ■

En gros, il s'agit de trouver, si c'est possible, un point dans un convexe qui maximise une fonction linéaire. Ce que l'on sait, c'est que la maximum, s'il existe, est atteint sur un sommet.

2 Formes normales pour les programmes linéaires

2.1 Programme canonique

Il est hyper naturel dans les applications d'exprimer des contraintes avec des inégalités. C'est très fréquent, par exemple les flots.

Définition 8 (programme canonique) Un programme canonique est de la forme :

$$\text{maximiser } c^t x$$
$$\begin{cases} Ax \leqslant b \\ x \geqslant 0 \end{cases}$$

où $c \in \mathbb{R}^d$, $A \in \mathfrak{M}_{m,d}(\mathbb{R})$ et $b \in \mathbb{R}^m$.

d = dimension = nombre de variables
m = nombre d'inégalités linéaires

L'exemple ci-dessus est un programme canonique. Le polyèdre des contraintes est formé par la contrainte $x \geq 0$ qui dit que l'on se trouve dans l'espace positif (quart du plan si on est en dimension 2). Chaque ligne de A dans $Ax \leq b$ vient dire que l'on se trouve d'un côté de l'hyperplan.

Exemple 9 Voici comment on écrit l'exemple avec une représentation matricielle :

$$\text{maximiser } x_1 + 6x_2 + 13x_3 \quad \begin{cases} x_1 \leqslant 200 \\ x_2 \leqslant 300 \\ x_1 + x_2 + x_3 \leqslant 400 \\ x_2 + 3x_3 \leqslant 600 \end{cases}$$

$$\text{maximiser } \begin{pmatrix} 1 & 6 & 13 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \begin{cases} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leqslant \begin{pmatrix} 200 \\ 300 \\ 400 \\ 600 \end{pmatrix} \end{cases}$$

Proposition 10 Tout programme linéaire peut s'écrire sous la forme d'un programme canonique équivalent.

Démonstration.

1. Passer de maximiser à minimiser (ou vice et versa) : multiplier la fonction objectif par -1.
2. Passer d'égalités à des inégalités : $a^T x = b$ devient $a^T x \leq b$ et $a^T x \geq b$
3. N'avoir que des variables positives : Remplacer x par $x^+ - x^-$ et ajouter $x^+, x^- \geq 0$.

■

2.2 Programme équationnel

Les inégalités c'est compliqué à manipuler d'un point de vue algorithmique. On le verra par la suite. L'algorithme du simplexe ne manipule que des contraintes qui sont des égalités.

Définition 11 (programme équationnel) Un programme équationnel est de la forme

$$\text{maximiser } c^t x \quad \begin{cases} Ax = b \\ x \geqslant 0 \end{cases}$$

où $c \in \mathbb{R}^n$, $A \in \mathfrak{M}_{m,n}(\mathbb{R})$ et $b \in \mathbb{R}^m$, où A est de rang m.

Pour convertir un programme canonique en programmation équationnel. Il faut convertir chaque inégalité en égalité. Pour chaque contrainte, par exemple $2x_1 + 3x_2 \leq 5$, on introduit une variable d'écart, disons x_{1000} qui vient remplacer l'écart entre $2x_1 + 3x_2$ et 5. La contrainte devient :

$$2x_1 + 3x_2 + x_{1000} = 5.$$

Comme les autres variables, les variables d'écart sont positives.

Proposition 12 Tout programme canonique se réécrit en un programme équationnel équivalent en introduisant des **variables d'écart** $x_{d+1}, \ldots, x_{d+m}$.

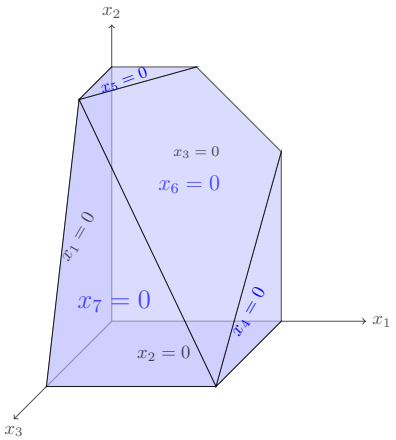
Typiquement, on a :

d = dimension
m = nombre d'inégalités linéaires
$n = d + m$ = nombre de variables en tout

Exemple 13 On introduit des **variables d'écart** x_4, x_5, x_6, x_7 de la façon suivante :

$$\text{maximiser } x_1 + 6x_2 + 13x_3 \quad \begin{cases} x_1 + x_4 = 200 \\ x_2 + x_5 = 300 \\ x_1 + x_2 + x_3 + x_6 = 400 \\ x_2 + 3x_3 + x_7 = 600 \end{cases}$$

Ici, $d = 3$ car le problème initial était sur trois variables $x_1, \ldots, x_3$. Il y a $m = 4$ contraintes et donc $m = 4$ variables d'écart. En tout, on a $3 + 4$ variables.



$\underbrace{x_1 \ldots x_d}_{d \text{ variables dans l'espace de départ}}$ $\underbrace{x_{d+1} \ldots x_{d+m}}_{m \text{ variables d'écart} = m \text{ contraintes}}$

Face = une équation $x_i \geq 0$

La forme équationnel est très pratique. Toutes les faces sont devenues des équations de la forme $x_i = 0$, que la face était juste un bord pour rester dans l'espace positif, ou alors une face issue d'une contrainte.

3 Algorithme du simplexe (avec les mains)

Un algorithme naïf pourrait être de calculer la fonction objectif en *tous* les sommets. Puis de choisir un meilleur sommet. Le soucis est qu'il peut y avoir un nombre exponentiel de sommets en le nombre de variables. Pensez à l'hypercube.

Au lieu de cela, nous allons nous *balader de sommets en sommets*. A chaque fois, on va vers un sommet qui améliore l'objectif. C'est l'algorithme du simplexe.

3.1 Idée générale

```
fonction simplexe
    sommet = sommet initial
    tant que sommet non optimal
        trouver un sommet voisin qui améliore l'objectif
    renvoyer sommet optimal
```

L'enjeu est maintenant de savoir comment représenter un sommet, et savoir tester si un sommet est optimal, et trouver un sommet voisin qui améliore l'objectif. Avant cela, nous allons discuter de représentation de programme linéaire. Car aussi, étonnant que cela puisse paraître, nous allons représenter un sommet un programme linéaire ! L'algorithme du simplexe va réécrire des programmes linéaires.

3.2 Représentation d'un sommet du polyèdre

Un sommet est à l'intersection de d faces. Comment donc représenter un sommet ? Tout simplement avec un ensemble de d faces. Si on considère que notre programme est équationnel, c'est la donnée de d variables mises à 0 :

$$x_{i_1} = 0; \dots x_{i_d} = 0.$$

Exemple 14 Par exemple, l'origine $(x_1, x_2, x_3) = (0, 0, 0)$ est un sommet du polyèdre dans l'exemple. Il est représenté par $x_1 = x_2 = x_3 = 0$.

Exemple 15 Le point $(x_1, x_2, x_3) = (200, 0, 0)$ est représenté par $x_4 = x_2 = x_3 = 0$.

3.3 Trouver un sommet voisin

Comment donc se déplacer d'un sommet à un voisin qui améliore l'objectif ? En fait, nous écrivons le programme linéaire équationnel en mettant les variables d'écart à gauche comme suit :

$$\text{maximiser } x_1 + 6x_2 + 13x_3$$
$$\begin{cases} x_4 &= 200 - x_1 \\ x_5 &= 300 - x_2 \\ x_6 &= 400 - x_1 - x_2 - x_3 \\ x_7 &= 600 - x_2 - 3x_3 \end{cases}$$

On cherche alors améliorer l'objectif. Pour cela, on regarde si on incrémentant une des variables x_1, x_2 ou x_3 on améliore l'objectif. On a l'embaras du choix : les trois variables font augmenter l'objectif. Une des heuristiques de choix peut être d'augmenter la variable avec le coefficient le plus grand : ici 13, donc on peut choisir d'augmenter x_3. Ici, on va choisir d'augmenter x_1 (pour des raisons qu'on verra après).

En augmentant x_1, comme toutes les variables sont positives, on voit que la première contrainte $x_4 = 200 - x_1$ nous dit que x_1 ne peut pas dépasser 200. La deuxième contrainte $x_5 = 300 - x_2$ ne limite pas du tout x_1, pareil pour la dernière contrainte. La troisième contrainte $x_6 = 400 - x_1 - x_2 - x_3$ limite x_1 à 400. On choisit alors la contrainte qui limite le plus x_1 afin de garantir que toutes les variables soient positives. On sélectionne donc la première contrainte :

$$x_4 = 200 - x_1.$$

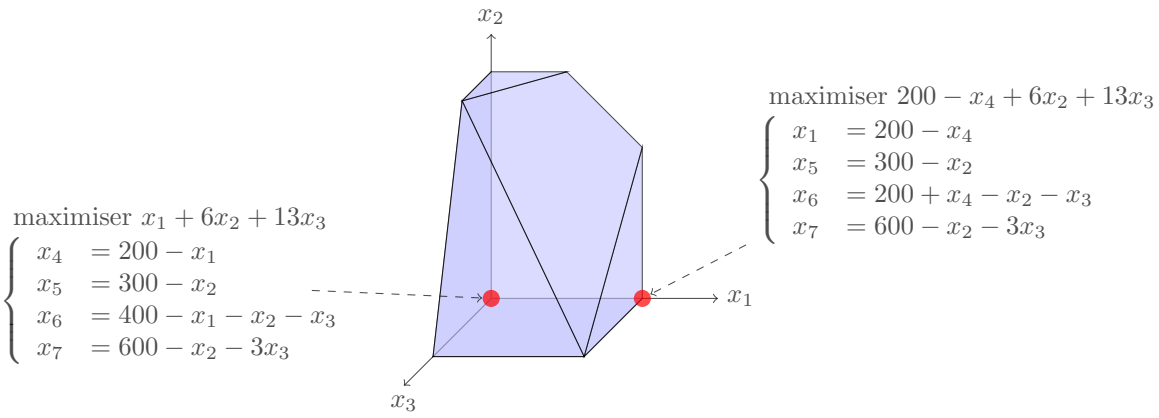
On réécrit la contrainte pour avoir $x_1 = ...$:

$$x_1 = 200 - x_4.$$

Maintenant, on réécrit l'objectif et les contraintes en remplaçant x_1 par $200 - x_4$. On obtient :

$$\text{maximiser } 200 - x_4 + 6x_2 + 13x_3$$
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 300 - x_2 \\ x_6 &= 200 + x_4 - x_2 - x_3 \\ x_7 &= 600 - x_2 - 3x_3 \end{cases}$$

Sur le polyèdre, à chaque sommet est attaché une 'réécriture' du programme équationnel initial : Le sommet $(x_1, x_2, x_3) = (0, 0, 0)$ est l'intersection des plans $x_1 = 0$, $x_2 = 0$ et $x_3 = 0$. Le sommet $(x_1, x_2, x_3) = (200, 0, 0)$ est lui l'intersection des plans $x_2 = 0$, $x_3 = 0$ et $x_4 = 0$. En ce sommet, la valeur de l'objectif est 200.



Définition 16 Un *tableau* est une telle réécriture du programmation équationnel initial.

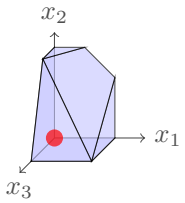
Intuitivement, les variables qui apparaissent dans l'objectif et dans la partie droite sont nulles.

Définition 17 (pivot) Le pivot est l'opération qui consiste à passer d'un tableau T à un autre :
- à choisir une variable x_i qui apparaît dans l'objectif de T avec un coefficient strictement positif ;
- à choisir la (une des) contrainte $x_j = ...$ qui limite le plus l'augmentation de x_i
- à réécrire la contrainte $x_j = ...$ comme une contrainte équivalente $x_i = ...$ dans le tableau
- à remplacer toutes les occurrences de x_i par son expression dans le reste du tableau T

3.4 Exemple d'exécution

Donnons l'exécution sur le programme suivant :

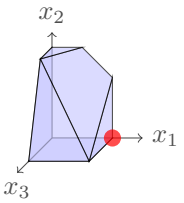
```
max x + 6y + 13z
x <= 200
y <= 300
x + y + z <= 400
y + 3z <= 600
```



maximiser $x_1 + 6x_2 + 13x_3$

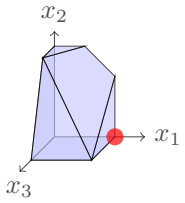
$$\begin{cases} x_4 &= 200 - x_1 \\ x_5 &= 300 - x_2 \\ x_6 &= 400 - x_1 - x_2 - x_3 \\ x_7 &= 600 - x_2 - 3x_3 \end{cases}$$

$x_4 := 0 \qquad x_1 \nearrow$



maximiser $200 - x_4 + 6x_2 + 13x_3$

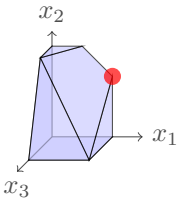
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 300 - x_2 \\ x_6 &= 200 + x_4 - x_2 - x_3 \\ x_7 &= 600 - x_2 - 3x_3 \end{cases}$$



maximiser $200 - x_4 + 6x_2 + 13x_3$

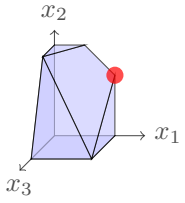
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 300 - x_2 \\ x_6 &= 200 + x_4 - x_2 - x_3 \\ x_7 &= 600 - x_2 - 3x_3 \end{cases}$$

$x_6 := 0 \qquad x_2 \nearrow$



maximiser $1400 + 5x_4 - 6x_6 + 7x_3$

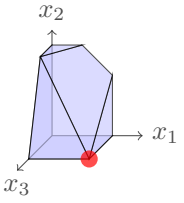
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 100 - x_4 + x_6 + x_3 \\ x_2 &= 200 + x_4 - x_6 - x_3 \\ x_7 &= 400 - x_4 + x_6 - 2x_3 \end{cases}$$



maximiser $1400 + 5x_4 - 6x_6 + 7x_3$

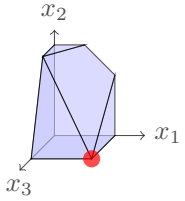
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 100 - x_4 + x_6 + x_3 \\ x_2 &= 200 + x_4 - x_6 - x_3 \\ x_7 &= 400 - x_4 + x_6 - 2x_3 \end{cases}$$

$x_2 := 0 \qquad x_3 \nearrow$



maximiser $2800 + 12x_4 - 13x_6 - 7x_2$

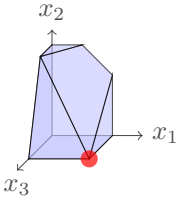
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 300 - x_2 \\ x_3 &= 200 + x_4 - x_6 - x_2 \\ x_7 &= -3x_4 + 3x_6 + 2x_2 \end{cases}$$



maximiser $2800 + 12x_4 - 13x_6 - 7x_2$

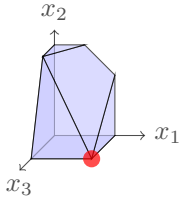
$$\begin{cases} x_1 &= 200 - x_4 \\ x_5 &= 300 - x_2 \\ x_3 &= 200 + x_4 - x_6 - x_2 \\ x_7 &= -3x_4 + 3x_6 + 2x_2 \end{cases}$$

$x_7 := 0 \qquad x_4 \nearrow$



maximiser $2800 - x_6 - 4x_7 + x_2$

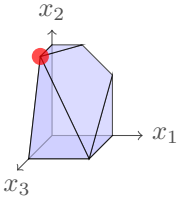
$$\begin{cases} x_1 &= 200 - x_6 + \frac{x_7}{3} - \frac{2x_2}{3} \\ x_5 &= 300 - x_2 \\ x_3 &= 200 - \frac{x_7}{3} - \frac{x_2}{3} \\ x_4 &= x_6 - \frac{x_7}{3} + \frac{2x_2}{3} \end{cases}$$



maximiser $2800 - x_6 - 4x_7 + x_2$

$$\begin{cases} x_1 &= 200 - x_6 + \frac{x_7}{3} - \frac{2x_2}{3} \\ x_5 &= 300 - x_2 \\ x_3 &= 200 - \frac{x_7}{3} - \frac{x_2}{3} \\ x_4 &= x_6 - \frac{x_7}{3} + \frac{2x_2}{3} \end{cases}$$

$x_5 := 0 \qquad x_2 \nearrow$



maximiser $3100 - 4x_7 - x_6 - x_5$

$$\begin{cases} x_1 &= 0 + \ldots \\ x_2 &= 300 - \epsilon_2 \\ x_3 &= 100 + \ldots \\ \epsilon_1 &= \ldots \end{cases}$$

3.5 Tester si un sommet est optimal

Un sommet est optimal si toutes les coefficients des variables dans l'objectif du tableau correspondant sont tous négatifs ou nuls. Par exemple dans

$$\begin{array}{l}\text{maximiser } 3100 - 4x_7 - x_6 - x_5 \\ \begin{cases} x_1 &= 0 + ... \\ x_2 &= 300 - \epsilon_2 \\ x_3 &= 100 + ... \\ \epsilon_1 &= ... \end{cases}\end{array}$$

les coefficients sont $-4, -1, -1$. Ils sont tous négatifs donc le sommet est optimal.

3.6 Dégénérescence

Dans l'exemple, il y a un moment où on reste au même sommet bien que le tableau change. Les variables de l'objectif et des parties droites sont nulles ; mais il y a aussi une variable x_j d'une contrainte $x_j = ...$ qui est nulle.

En 3D, un sommet est l'intersection de 3 plans. Une dégénérescence correspond à un sommet qui est intersection de strictement plus de 3 plans. Et oui : il y a les trois variables des parties droites, et x_j (dans le cas général de strictement plus de d hyper-plans).

4 Vocabulaire

Dans cette section, nous introduisons le vocabulaire utilisé en programmation linéaire.

4.1 Tableau

On a déjà vu la notion de tableau. Mais voici la définition formelle.

Définition 18 (tableau) Étant donné une partition $B \sqcup N = \{1, \dots, n\}$ avec $|N| = d$,‘ un tableau est un programme linéaire sous la forme

$$\begin{array}{l}\text{maximiser } v + c'^t x_N \\ \begin{cases} x_B = p - A' x_N \\ x \geqslant 0 \end{cases}\end{array}$$

où $v \in \mathbb{R}$, $c' \in \mathbb{R}^d$, $p \in \mathbb{R}^m$, $A' \in \mathfrak{M}_{m,d}(\mathbb{R})$ et le vecteur $x \in \mathbb{R}^n$ est décomposé en $x_B \in \mathbb{R}^m$ et $x_N \in \mathbb{R}^d$.

Dans un tableau, x_N est l'ensemble des variables qui sont censé valoir 0 (bien qu'elles ne valent pas réellement 0, le tableau ne le spécifie pas). Ce sont les variables qui apparaissent en partie droite des contraintes et aussi dans l'objectif.

La valeur v est la valeur courante de la fonction objectif en le sommet spécifié par $x_N = 0$. Les variables x_B sont les autres variables, que l'on exprime en fonction des variables x_N. Elles correspondent aux contraintes du tableau.

4.2 Solution basique

Le sommet (plus précisément les valeurs de toutes les variables, y compris les variables d'écart que l'on a ajouté, tout le monde est le bienvenu !) correspond à ce qu'on appelle *solution basique* du tableau.

Définition 19 (solution basique) Un tableau admet une solution basique si $p \geq 0$. La solution basique est le vecteur $x \in \mathbb{R}^n$ défini par $x_B = p$ et $x_N = 0$.

Étant donné un tableau, les valeurs des variables $x_1, \dots, x_d$ dans la solution basique d'un tableau donne le sommet que représente le tableau.

4.3 Variables de base et hors base

Définition 20 (variables de base) Une base B est un sous-ensemble de $\{1, \dots, n\}$ de cardinal m telle que A_B soit de rang m. Les variables dans x_B sont les variables de base.

On discutera dans 2 minutes de lecture pourquoi A_B doit être de rang m... juste dans la prochaine section.

Définition 21 (variables hors base) On note $N = \{1, \dots, n\} \setminus B$. Les variables dans x_N sont hors base.

tableau	= sommet à l'intersection des $x_i = 0$ où $i \in N$
v	= valeur de la fonction objectif en ce sommet
p	= valeurs des variables x_B de base en ce sommet

4.4 A toute base, son unique tableau

Pourquoi donc on demande à ce que A_B soit de rang m ? Qu'est ce que cela signifie donc ?

Dans un programme équationnel, il y a les contraintes qui sont $Ax = b$. Ces contraintes veulent dire que les variables sont *toutes* des variables d'écart : x_i est l'écart à la face i. Et $x \geq 0$ veut que toutes ces variables sont positives, i.e. on est *dans* le polyèdre (dans l'espace des solutions).

L'équation $Ax = b$ s'écrit $(A_N A_B)\begin{pmatrix} x_N \\ x_B \end{pmatrix} = b$. Mais quand A_B est de rang m, cela signifie donc que A_B est inversible, et que l'on peut écrire x_B en fonction de x_N. C'est dans la démonstration de la proposition suivante qui donne l'expression de l'unique (!) tableau dont la base est B.

Quand est-ce que x_B peut s'écrire en fonction de x_N. Exactement $x_N = 0$ donne un point, i.e. quand l'intersection des faces de N forme un sommet. Dans ce cas, les distances de ce sommet aux autres faces de B est complétement déterminé.

Au contraire, si l'intersection des faces de N ne formait pas un sommet, mais une droite, ou un plan, etc. les écarts aux autres faces n'est pas entièrement déterminé. Si on se balade dans l'intersection – disons une droite – les écarts aux faces changent. Heureusement, ce cas n'arrive jamais, car l'algorithme du simplexe se balade de sommet en sommet.

Proposition 22 On considère le programme équationnel :

$$\begin{array}{l} \text{maximiser } c^t x \\ \left\{ \begin{array}{l} Ax = b \\ x \geqslant 0 \end{array} \right. \end{array} \qquad \text{où } c \in \mathbb{R}^n,\ A \in \mathfrak{M}_{m,n}(\mathbb{R}) \text{ et } b \in \mathbb{R}^m, \text{ où } \underline{A \text{ est de rang } m}.$$

Étant donné une base B, il y a un unique tableau équivalent au programme équationnel. C'est :

$$\begin{array}{l} \text{maximiser } v + c'^t x_N \\ \left\{ \begin{array}{l} x_B = p - A' x_N \\ x \geqslant 0 \end{array} \right. \end{array} \qquad \text{avec } v = c_B^t A_B^{-1} b\,;\ c' = c_N - (c_B^t A_B^{-1} A_N)^t\,;\ p = A_B^{-1} b\,;\text{ et } A' = A_B^{-1} A_N.$$

Démonstration. On note $N = \{1, \dots, n\} \setminus B$. Le système de contraintes $Ax = b$ se décompose en

$$A_N x_N + A_B x_B = b.$$

Comme B est une base, A_B est de rang m, donc on peut exprimer x_B en fonction du reste :

$$x_B = A_B^{-1} b - A_B^{-1} A_N x_N.$$

On obtient donc les contraintes du tableau.

Pour l'objectif, il faut réécrire :

$$\begin{aligned} c^t x &= c_B^t x_B + c_N^t x_N \\ &= c_B^t (A_B^{-1} b - A_B^{-1} A_N x_N) + c_N^t x_N \\ &= c_B^t (A_B^{-1} b + (c_N - (c_B^t A_B^{-1} A_N)^t) x_N \end{aligned}$$

Bon, ça y est, on a l'existence. Maintenant montrons l'unicité. Supposons qu'il y en ait deux tableaux pour la même base :

$$\begin{array}{l} \text{maximiser } v + c'^t x_N \\ \left\{ \begin{array}{l} x_B = p - A' x_N \\ x \geqslant 0 \end{array} \right. \end{array} \qquad \text{et} \qquad \begin{array}{l} \text{maximiser } v' + c''^t x_N \\ \left\{ \begin{array}{l} x_B = p' - A'' x_N \\ x \geqslant 0 \end{array} \right. \end{array}$$

$x_B = p - A' x_N$ et $x_B = p' - A'' x_N$ sont deux relations qui donnent les valeurs des écarts aux faces B en connaissant les écarts aux faces N. Ces deux relations sont indépendantes de $x \geq 0$ qui dit juste que l'on est 'à l'intérieur ou sur le bord du polyèdre'.

Si $x_N = 0$, on obtient $p = p'$. Si on prend $x_N =$ un vecteur de base canonique $e_j = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, alors $x_B = p - A' e_j =$ $p - A'' e_j$ donne $A' e_j = A'' e_j$. Cela signifie que la colonne numéro j de A' est égale à la colonne numéro j de A''. Autrement dit, on obtient $A' = A''$.

De même on obtient $v = v'$ et $c' = c''$. ■

Exemple 1 Considérons le programme équationnel $\begin{array}{l} \text{maximiser } x_1 + 6x_2 + 13x_3 \\ \begin{cases} x_1 + x_4 & = 200 \\ x_2 + x_5 & = 300 \\ x_1 + x_2 + x_3 + x_6 & = 400 \\ x_2 + 3x_3 + x_7 & = 600 \end{cases} \end{array}$

Pour la base $B = \{4, 5, 6, 7\}$, on a $A_B = Id_4$, $c_B^t = (0, 0, 0, 0)$ et $c_N^t = (1, 6, 13)$.

Pour la base $B = \{1, 5, 6, 7\}$, on a $A_B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$, $c_B = (1, 0, 0, 0)$ et $c_N = (6, 13, 0)$.

Exemple 23 Considérons un espace de solution qui est le carré $[0, 1]^2$. Par exemple :

$\begin{array}{l} \text{maximiser } x_1 + x_2 \\ \begin{cases} x_1 + x_3 \leq 1 \\ x_2 + x_4 \leq 1 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases} \end{array}$

On a $A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$.

Quand on met dans N exactement les variables de faces qui s'intersectent en 1 point, A_B est inversible.

Exemple 24 Considérons à nouveau un espace de solution qui est le carré $[0, 1]^2$, mais décrit avec en plus une contrainte inutile. Par exemple :

$\begin{array}{l} \text{maximiser } x_1 + x_2 \\ \begin{cases} x_1 + x_3 \leq 1 \\ x_2 + x_4 \leq 1 \\ x_1 + x_5 \leq 2 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases} \end{array}$

On a $A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$.

Si on prend $N = \{3, 5\}$, autrement dit les contraintes $x_1 \leq 1$ et $x_1 \leq 2$, alors la matrice A_B est $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ n'est pas inversible.

5 Algorithme du simplexe (plus formel)

5.1 Pseudo-code

entrée : un tableau τ admettant une solution basique
sortie : le maximum ou alors lève une exception non borné
fonction simplexe(τ)
    tant que il y a un coefficient strictement positif dans la fonction objectif de τ
        $\tau = \text{pivot}(\tau)$
    renvoyer la solution basique de τ

entrée : un tableau τ admettant une solution basique
sortie : un tableau équivalent, admettant une solution basique, améliorant non strictement l'objectif, de même espace de solutions, et ou alors lève une exception non borné
fonction pivot(τ)
    renvoyer le tableau obtenu à partir de τ comme suit.
    1. repérer une variable x_e avec coefficient strictement positif dans la fonction objectif
    2. repérer la contrainte $x_s = \dots$ qui limite le plus la croissance de x_e
       s'il n'y a pas de telle contrainte **raise** non borné
    3. augmenter x_e jusqu'à ce que $x_s := 0$ $x_e := 0$ $x_s \nearrow$

fonction *pivot* $\left(\begin{array}{l}\textbf{maximiser } v + c^t x_N \\ \begin{cases} x_B = b - A x_N \\ x \geq 0 \end{cases}\end{array}\right)$

choisir $j_0 \in N$ tel que $c_{j_0} > 0$

choisir $i_0 \in B$ tel que $-a_{i_0,j_0} < 0$ et $\frac{b_{i_0}}{a_{i_0,j_0}}$ minimal, s'il n'y a pas, **raise** "non borné"

renvoyer $\begin{array}{l}\textbf{maximiser } v' +^t c' x_{N'} \\ \begin{cases} x_{B'} = b' - A' x_{N'} \\ x \geq 0 \end{cases}\end{array}$ où pour tout $i \in B$, $j \in N$:

- $N' = \{i_0\} \cup N \setminus \{j\}$;
- $B' = \{j_0\} \cup B \setminus \{i\}$;
- $v' = v + \frac{b_{i_0} c_{j_0}}{a_{i_0,j_0}}$
- $c'_j = c_j - c_{j_0} \frac{a_{i_0,j}}{a_{i_0,j_0}}$
- $c'_{i_0} = \frac{-c_{j_0}}{a_{i_0,j_0}}$
- $b'_i = b_i - a_{i,j_0} \frac{b_{i_0}}{a_{i_0,j_0}}$
- $b'_{j_0} = \frac{b_{i_0}}{a_{i_0,j_0}}$
- $a'_{i,j} = a_{i,j} - \frac{a_{i,j_0} a_{i_0,j}}{a_{i_0,j_0}}$
- $a'_{i,i_0} = \frac{a_{i,j_0}}{a_{i_0,j_0}}$
- $a'_{j_0,i_0} = \frac{1}{a_{i_0,j_0}}$
- $a'_{j_0,j} = \frac{-a_{i_0,j}}{a_{i_0,j_0}}$.

6 Correction

Proposition 25 (admis car lourd) La fonction pivot est correct.

Théorème 26 L'algorithme du simplexe, s'il termine, retourne bien la valeur maximale de l'objectif.

Démonstration.

$$\begin{array}{l}\text{maximiser } v + c'^t x_N \\ \begin{cases} x_B = p - A' x_N \\ x \geqslant 0 \end{cases}\end{array}$$

L'algorithme termine donc les coefficients de c' sont tous négatifs. Sur l'espace des solutions, la valeur v majore l'objectif. La valeur v est atteinte pour la solution basique, qui est donc solution optimale. ■

7 Terminaison de l'algorithme du simplexe

L'algorithme du simplexe peut malheureusement boucler. Voici un exemple qui est donné p. 29 dans [Van98] :

```
max x - 2y -2t
0.5x - 3.5y - 2z + 4t <= 0
0.5x - y - 0.5z + 0.5t <= 0
x <= 1
```

L'exemple boucle avec la règle suivante : faire entrer dans la base la variable avec le plus grand coefficient dans l'objectif ; faire sortir celle avec la plus limitante avec le plus indice.

demo

En fait, on se fixe un ordre total sur les noms des variables, typiquement on utilise l'indice. Si jamais il n'y a pas d'indices, c'est juste un ordre total fixé.

Définition 27 (règle de Bland) La règle de Bland est la stratégie consistant à choisir la variable entrante candidate d'indice minimal, et la variable sortante candidate d'indice minimal.

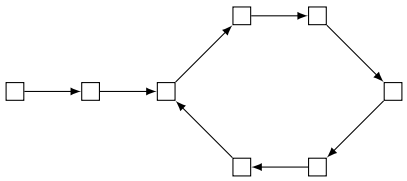
C'est comme si l'ordre total donnait une priorité aux variables. La règle de Bland consiste à choisir à chaque fois la variable la plus prioritaire. La notion de priorité n'évolue pas dans le temps. Avec cette règle, l'algorithme du simplexe termine sur l'exemple. En fait, il termine toujours avec la règle de Bland.

Proposition 28 (terminaison) Si on applique toujours la règle de Bland, l'algorithme du simplexe termine.

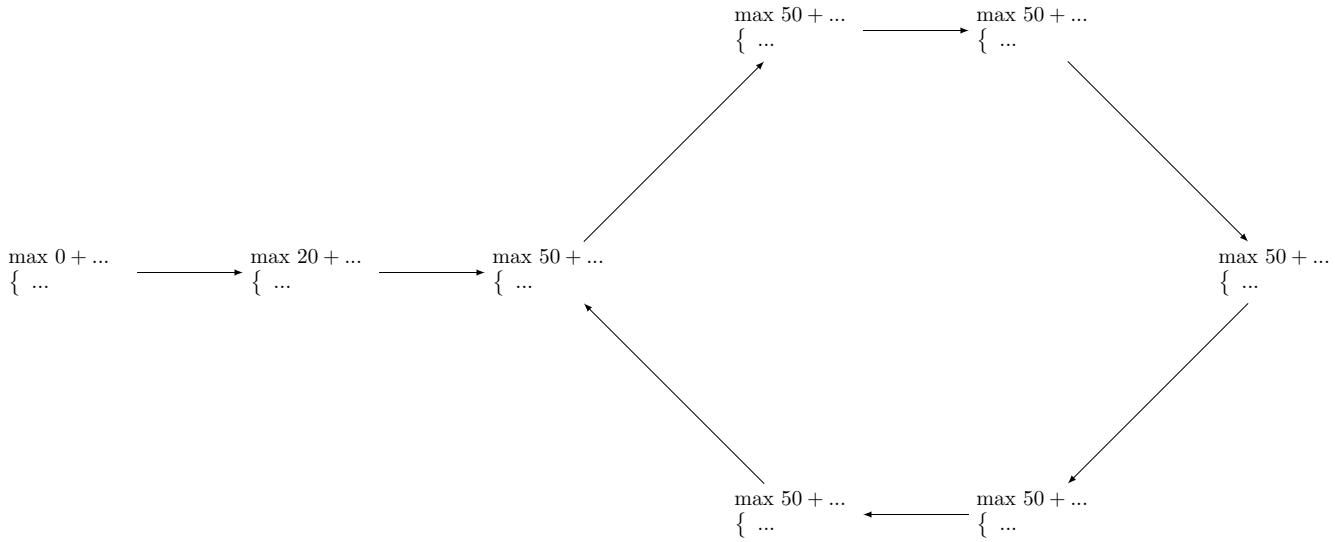
Démonstration. Par l'absurde. Supposons que l'algorithme ne termine pas sur le programme équationnel initial suivant :

$$\begin{array}{l}\text{maximiser } c^t x \\ \begin{cases} Ax = b \\ x \geq 0 \end{cases}\end{array}$$

Chaque tableau est caractérisé par les variables qui sont dans la base, puisque les variables en dehors, sont pensées comme nulles (c'est la solution basique qui donne le sommet). Ainsi, l'ensemble des tableaux est fini. Nous avons donc un cycle dans l'exécution : l'algorithme du simplexe revient sur le même tableau. Dans le dessin suivant, chaque point représente un tableau et une flèche représente un pivotage.



L'exécution commence du tableau initial puis on atteint un tableau sur lequel on boucle. N'oublions que la valeur de l'objectif courante ne fait qu'augmenter. Comme on boucle la valeur de l'objectif stagne le long du cycle. Dans le graphe suivant, la valeur de l'objectif stagne à 50.



Soit F les variables manipulées par un pivotage pendant le cycle. Chaque variable de F entrent à un moment dans la base, puis en ressortent car on doit retomber sur la même base durant le cycle. Les variables correspondantes sont dites "folles" (mais appelées aussi "capricieuses" dans la littérature).

Fait 29 Les variables "folles" sont nulles dans les solutions basiques du cycle.

Démonstration.

On montre d'abord que toute variable folle x_u vaut 0 dans la solution basique dans un tableau T où elle sort de la base.

> Considérons une variable x_u folle. Considérons un tableau T où x_u sort de la base dans le cycle. Par l'absurde, si sa valeur dans la solution basique était strictement positif, alors l'objectif augmenterait strictement dans le cycle. Ce n'est pas possible. Donc dans ce tableau, x_u vaut 0 dans la solution basique.

Soit x_u une variable folle. Soit T_0 un tableau où u sort. Soit T_n le tableau obtenu depuis T_0 après n pivots.
Soit la propriété :

$$\mathcal{P}(n) : x_u \text{ vaut 0 dans la solution basique dans } T_n.$$

$\mathcal{P}(0)$ est vraie par ce qu'on vient de voir.
Supposons $\mathcal{P}(n)$ montrons $\mathcal{P}(n+1)$.

— Si x_u sort de la base, alors x_u vaut 0 car hors base.
— Si x_u rentre dans la base, elle pivote avec une autre variable folle x_v qui est dans la base et qui va en sortir. Mais alors v vaut 0 dans la solution basique de T_n. En T_n on a une contrainte :

$$x_v = 0 + ... - 5x_u + ...$$

Ainsi, le pivot fait que x_u continue à valoir 0 dans la solution basique de T_{n+1} où il y a :

$$x_u = 0 + ...\dot{}.$$

■

Fait 30 Toutes les solutions basiques sont égales le long du cycle.

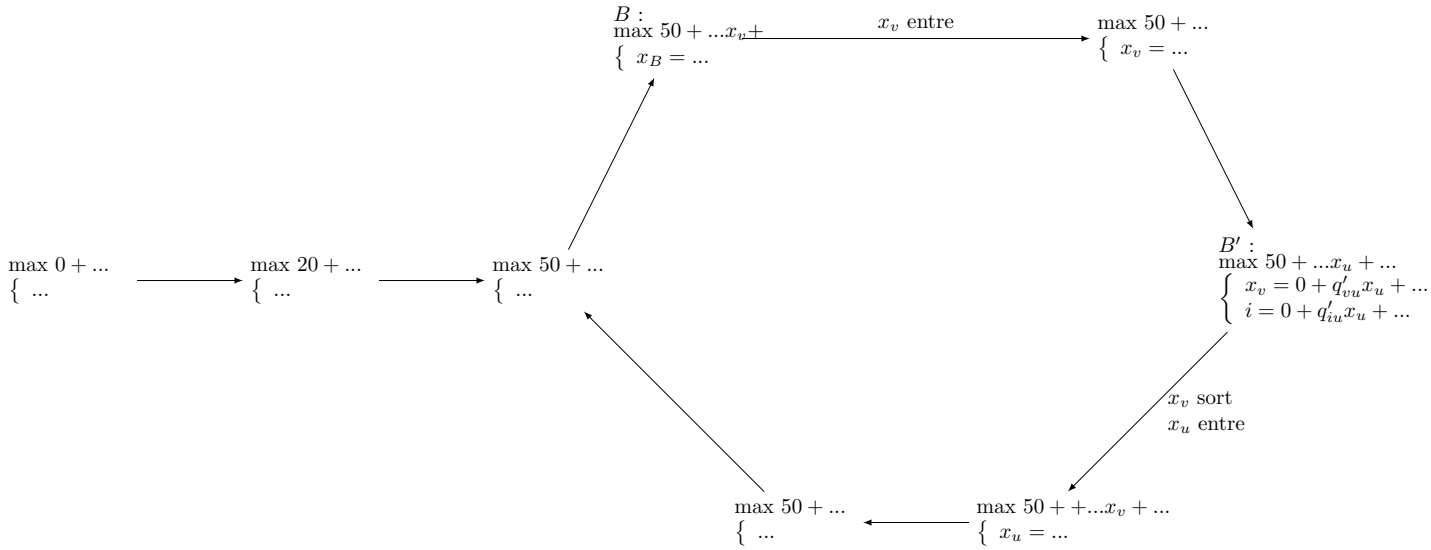
Démonstration.

— On vient de voir que les variables folles sont nulles.
— On a $N \setminus F = N' \setminus F$, puisque F est exactement l'ensemble des variables folles. L'ensemble $N \setminus F$ est l'ensemble des variables qui restent hors base le long du cycle. Pour $i \in N \setminus F$, x_i est nulle dans une solution basique.

— On a $B \setminus F = B' \setminus F$. Pour $i \in B \setminus F$, c'est la contrainte $x_i = valeur + ...$ qui donne la valeur de x_i. Cette valeur ne change pas puisque l'on fait toujours des remplacements avec des variables folles qui ont une valeur nulle, même quand elles sont dans la base.

■

Soit x_v la variable dans F la moins prioritaire (i.e. d'indice le plus grand). On considère maintenant une base B du cycle où le pivotage qui suit fait entrer v dans la base. On considère aussi une base B' du cycle où le pivotage qui suit fait sortir v, et qui fait entrer une certaine variable d'indice que l'on note u.



Afin de ne pas alourdir la démonstration, nous ferons des abus de langage. On dira : "en B" pour dire "dans le tableau correspondant à la base B", "dans l'objectif de B" pour dire "dans la fonction objectif écrite comme $z_0 + cx_N$ dans le tableau de la base B, où $N = \{1, \ldots, n\} \setminus B$".

Commençons par deux faits qui découlent de l'utilisation de la règle de Bland.

Fait 31 Dans l'objectif de B, les coefficients devant les variables de $F \cap N \setminus \{v\}$ sont ≤ 0, le coefficient devant x_v est strictement positif.

Démonstration.

$$B : \begin{array}{l} \max\ 50 + ...x_v + ... \\ \{\ x_B = ... \end{array}$$

La variable x_v entre dans la base, donc son coefficient dans l'objectif est strictement positif.

Par l'absurde, si le coefficient devant x_f où $f \in F \cap N \setminus \{v\}$ était strictemnt positif,

$$B : \begin{array}{l} \max\ 50 + ...x_v + ... + 3x_f + ... \\ \{\ x_B = ... \end{array}$$

alors x_v n'aurait pas été sélectionnée pour entrer dans la base, car en appliquant la règle de Bland, on aurait pris une variable plus prioritaire que x_v. Contradiction car x_v est sélectionnée. Donc le coefficient devant x_f est ≤ 0.

■

Fait 32 En B', le coefficient devant x_u dans la contrainte $x_v = ...$ est < 0. Pour tout $i \in F \cup B' \setminus \{v\}$, le coefficient devant x_u dans la contrainte $x_i = ...$ est ≥ 0.

Formellement, en B', on a $q'_{vu} < 0$, et pour tout $i \in F \cup B' \setminus \{v\}$, on a $q'_{iu} \geq 0$.

Démonstration. Le raisonnement est similaire mais concerne le choix de v à faire sortir de la base B'.

$$B' : \begin{array}{l} \max\ 50 + ...x_u + ... \\ \left\{ \begin{array}{l} x_v = 0 + q'_{vu} x_u + ... \\ i = 0 + q'_{iu} x_u + ... \end{array} \right. \end{array}$$

Primo, v est candidate pour sortir et donc $q'_{vu} < 0$ (on simule une "croissance" de 0 de x_u qui entre dans la base). Deuxio, soit $i \in F \cup B' \setminus \{v\}$. Par l'absurde, supposons $q'_{iu} < 0$. Alors i est aussi candidate pour sortir de la base ; et elle est plus prioritaire. Donc x_v n'aurait pas sélectionnée pour sortir de la base avec la règle de Bland. Contradiction. Donc $q'_{iu} \geq 0$. ■

Reconsidérons le programme initial :

$$\begin{array}{l}\text{maximiser } c^t x \\ \begin{cases} Ax = b \\ x \geq 0 \end{cases}\end{array}$$

On supprime les contraintes $x \geq 0$ (qui disent que l'on est à l'intérieur du polyèdre) que l'on remplace par les trois contraintes suivantes :

$$\begin{aligned} x_{F\setminus\{v\}} \geq 0 &\text{ par rapport aux faces } F \setminus \{v\} \text{ on est du côté du polyèdre} \\ x_v \leq 0 &\text{ on est à l'extérieur de la face } v \\ x_{N\setminus F} = 0 &\text{ on est sur les faces } x_{N\setminus F} \end{aligned}$$

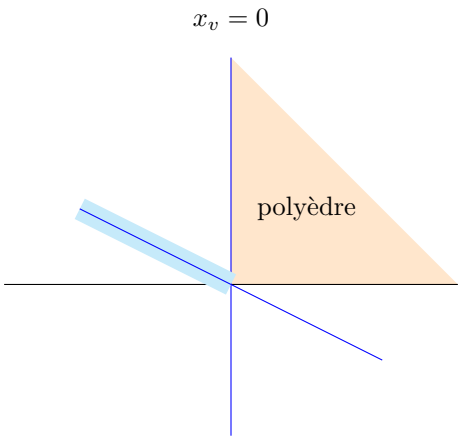
où $N = \{1, \ldots, n\} \setminus B$.

Les contraintes de positivité ou négativité pour x_B ne sont plus données explicitement, c'est $Ax = b$ qui fait le travail.

On obtient donc le programme linéaire auxiliaire suivant :

$$\begin{array}{l}\text{maximiser } c^t x \\ \begin{cases} Ax = b \\ x_{F\setminus\{v\}} \geq 0 \\ x_v \leq 0 \\ x_{N\setminus F} = 0 \end{cases}\end{array}$$

En image, cela donne ça, avec en trait bleu les faces N (dont $x_v = 0$) :



La ligne bleue ciel est la zone délimitée par les trois nouvelles inégalités : $x_v \leq 0$ dit qu'on est à gauche de $x_v = 0$ (de l'autre côté du polyèdre), mais $x_{N\mathsf{F}} = 0$ dit que l'on reste sur la ligne bleu ciel.

On montre deux faits contradictoires : que ce programme linéaire admet un optimum, mais qu'il est aussi non borné (on est capable de rendre l'objectif aussi grand que l'on souhaite). Ainsi, on aboutit à une contradiction. Ce qui montre que l'algorithme du simplexe termine. Terminons sur ces deux faits contradictoires.

Fait 33 Le programme linéaire auxiliaire admet un optimum.

Démonstration. Soit $\tilde{x}$ la solution basique en B. Montrons que $\tilde{x}$ est un optimum.

1) Montrons d'abord que c'est une solution du programme auxiliaire. Tout d'abord, c'est une solution du programmation équationnel donc on a $Ax = b$. Maintenant, il suffit de vérifier que les trois contraintes supplémentaires sont vraies :

$\tilde{x}_{F\setminus\{v\}} \geq 0$: ok car ces variables sont nulles (cf. Fait 29)
$\tilde{x}_v \leq 0$: Comme $v \in F$, pareil par le fait 29, $x_v = 0$
$\tilde{x}_{N\setminus F} = 0$: car $\tilde{x}_N = 0$ car $\tilde{x}$ est basique en B.

2) Il reste à montrer que $\tilde{x}$ est une solution *optimale*. Pour cela, on regarde comment l'objectif en B évolue si on modifie les valeurs des variables x_N. La table suivante résume la situation :

Variables	Contraintes	Coefficients	Possibilité d'augmenter l'objectif
$x_{N\setminus F}$	$x_{N\setminus F} = 0$	on s'en fiche	non
$x_{F\setminus\{v\}}$	$x_{F\setminus\{v\}} \geq 0$	négatif	non
x_v	$x_v \leq 0$	positif	non

— Comme il est contraint que $x_{N\setminus F} = 0$, on ne peut pas modifier ces variables : elles sont nulles à tout jamais. On ne peut pas les modifier. Donc les coefficient devant ces dernières n'ont aucune importance.

— D'après le fait 31, les coefficients devant $F \setminus \{v\}$ sont négatifs. Comme on a la contrainte $x_{F\setminus\{v\}} \geq 0$, si on modifie ces variables, on va diminuer l'objectif.
— En B, l'algorithme du simplexe fait entrer la variable v dans la prochaine base. Donc le coefficient est positif. Mais comme on a maintenant la contrainte $x_v \leq 0$ dans le programme auxiliaire, si on modifie x_v, on va diminuer l'objectif.

Quelque soit la façon dont on modifie $\tilde{x}$, l'objectif diminue. C'est un maximum local donc un maximum global (c'est comme ça en optimisation linéaire). Bref, $\tilde{x}$ est une solution optimale du programme linéaire auxiliaire.
■

Fait 34 L'objectif du programme linéaire auxiliaire est non borné.

Démonstration. Rappel :

$$B : \begin{cases} \max 50 + ...x_v + \\ x_B = ... \end{cases} \qquad B' : \begin{cases} \max 50 + ...x_u + ... \\ x_v = 0 + q'_{vu}x_u + ... \\ i = 0 + q'_{iu}x_u + ... \end{cases}$$

Considérons $\tilde{x}$ solution basique en B. Par le fait 30, c'est la solution basique en B' aussi.
Définissons la solution potentielle suivante, paramétrée par $t \geq 0$:

$$\begin{cases} \tilde{x}(t)_u = t \text{ pour la coordonnée } u \in N' \text{ (qui va entrer dans la base depuis } B') \\ \tilde{x}(t)_i = 0 \text{ pour tout } i \in N' \setminus \{u\} \\ \tilde{x}(t)_{B'} = p' + Q'\tilde{x}(t)_{N'}. \end{cases}$$

Autrement dit, toutes les coordonnées de $\tilde{x}(t)_{N'}$ sont nulles, sauf celles de u qui vaut t et donc qui peut grandir arbitrairement. Puis, $\tilde{x}(t)_{B'}$ est définie par le système d'équation en B' : $x_{B'} = p' + Q'x_{N'}$.
1) Montrons que $\tilde{x}(t)$ est dans l'espace de solutions du programme auxiliaire.
$\boxed{Ax = b}$ Tout d'abord, $A\tilde{x}(t) = b$ comme conséquence de $x_{B'} = p' + Q'x_{N'}$..
$\boxed{x_{N\setminus F} = 0}$ On a $N' \setminus F = N \setminus F$, par définition F, car seulement les variables de F se promènent. Donc $\tilde{x}(t)_{N\setminus F} = \tilde{x}(t)_{N'\setminus F} = 0$.
$\boxed{x_v \leq 0.}$ Comme x_v est une variable capricieuse, $\tilde{x}_v = 0$. Comme x_v sort en B', d'après le fait 32, $q'_{vu} < 0$. Ainsi, en lisant la ligne $\tilde{x}(t)_v = ...$ on a :

$$\tilde{x}(t)_v = \tilde{x}_v + tq'_{vu} < 0.$$

$\boxed{x_{F\setminus\{v\}} \geq 0.}$ Pour tout $i \in F \setminus \{v\}$, on a aussi $\tilde{x}_i = 0$ car i est capricieuse. Si $i \in F \cap N'$, alors $\tilde{x}(t)_i = 0$. Sinon, si $i \in B' \cap F \setminus \{v\}$, par le fait 32, $q'_{iu} \geq 0$. Et donc :

$$\tilde{x}(t)_i = \tilde{x}_i + tq'_{iu} \geq 0.$$

2) Pour finir, montrons la fonction objectif n'est pas bornée. Comme x_u est candidate pour entrer dans la base B', le coefficient devant x_u dans la fonction objectif du tableau B' est strictement positif et donc :

$$c^t\tilde{x}(t) = z'_0 + t \times \text{ce coefficient positif} \xrightarrow[t\to+\infty]{} +\infty.$$

La fonction objectif n'est pas bornée. ■
■

Remarque 35 En pratique, la règle de Bland est lente. On préfère perturber un peu b pour supprimer les dégénérescences. ([DPV08], p. 218)

8 Complexité

On note n le nombre de variables en tout (variables de départ + variables d'écart). Bref, le nombre de variables dans le programmation équationnel. On note d le nombre de variables initiales dans le programme canonique (ou aussi $d = |N|$).

Théorème 36 L'algorithme du simplexe réalise au plus $\binom{n}{d}$ itérations.

Démonstration. On sait que l'algorithme termine avec la règle de Bland. Il ne visite donc pas le même tableau plusieurs fois. Un tableau est caractérisé par les m variables dans la base (ou par les d variables dans N) parmi les n variables en tout. Il y a donc $\binom{n}{d}$ tableaux possibles. ■
On trouve sur Wikipedia `https://en.wikipedia.org/wiki/Klee\OT1\textendashMinty_cube` un programme où l'algorithme du simplexe met un temps exponentiel. C'est l'exemple de Klee-Minty, qui est un hypercube déformé. Plusieurs règles prennent un temps exponentiel.

8.1 Avec la règle de meilleur coeff

```
max 4x1 + 2x2 + x3
x1 <= 1
4x1 + x2 <= 100
8x1 + 4x2 + x3 <= 10000
```

Plus généralement,

$$\begin{array}{l}\text{maximiser } \sum_{j=1}^{n} 2^{n-j}x_j \\ \begin{cases} 2\sum_{j=1}^{i-1} 2^{i-j}x_j + x_i \leq 100^{i-1} \\ x_1, x_2, \ldots, x_n \geq 0 \end{cases}\end{array}$$

8.2 Pour la règle de Bland

Dans une note de D. Avis et V. Chvátal [] TODO: à compléter, ils discutent de la règle de Bland et que elle aussi donne une complexité exponentielle dans le pire cas.

Nous allons déformer légèrement le programme linéaire suivant dont la région est un hyper-cube de dimension n :

$$\begin{array}{l}\text{maximiser } x_n \\ \begin{cases} x_1 \leq 1 \\ x_2 \leq 1 \\ x_3 \leq 1 \\ \vdots \\ x_n \leq 1 \\ x_1, x_2, \ldots, x_n \geq 0 \end{cases}\end{array}$$

en introduisant un paramètre $\epsilon \in]0, 1/2[$. Le programme linéaire de Klee-Minty est le suivant :

$$\begin{array}{l}\text{maximiser } \epsilon^{n-1}x_1 + \epsilon^{n-2}x_2 + \ldots \epsilon x_{n-1} + x_n \\ \begin{cases} x_1 \leq 1 \\ 2\epsilon x_1 + x_2 \leq 1 \\ 2\epsilon^2 x_1 + 2\epsilon x_2 + x_3 \leq 1 \\ \vdots \\ 2\epsilon^{n-1}x_1 + 2\epsilon^{n-2}x_2 + \ldots 2\epsilon x_{n-1} + x_n \leq 1 \\ x_1, x_2, \ldots, x_n \geq 0 \end{cases}\end{array}$$

Pour $n = 3$, voici le code pour $\epsilon = 0.1$:

```
max 0.001x1 + 0.1x2 + x3
x1 <= 1
0.2x1 + x2 <= 1
0.002x1 + 0.2x2 + x3 <= 1
```

Pour $n = 4$, voici le code pour $\epsilon = 0.1$:

```
max 0.001x1 + 0.01x2 + 0.1x3 + x4
x1 <= 1
0.2x1 + x2 <= 1
0.02x1 + 0.2x2 + x3 <= 1
0.002x1 + 0.02x2 + 0.2x3 + x4 <= 1
```

9 Prétraitement : obtenir tableau avec solution basique

entrée : un programme canonique L
sortie : insatisfaisable si les contraintes de L sont insatisfaisables ou un tableau avec solution basique de même maximum que L

fonction prétraitement $\left(L : \begin{cases} \text{maximiser } c^t x \\ Ax \leqslant b \\ x \geqslant 0 \end{cases} \right)$

si $b \geq 0$
renvoyer L //youpi, L admet directement la solution nulle comme solution basique

soit le programme $L_{aux} : \begin{cases} \text{maximiser } -x_0 \\ Ax - x_0 \leqslant b \quad \text{où } x_0 \text{ est une variable auxiliaire fraîche} \\ x_0, x \geqslant 0 \end{cases}$

//l'espace des solutions de L_{aux} est non vide
//l'espace des solutions de L est non vide ssi l'optimum de L_{aux} est 0.

1. mettre L_{aux} sous forme équationnelle en introduisant des variables d'écart, puis sous tableau
2. faire le pivot qui fait entrer x_0 dans la base, et sortir la variable x_k où b_k minimum

//on réalise ce pivot même si le coefficient devant x_0 est négatif (-1)
//le tableau obtenu admet une solution basique

3. Lancer l'algorithme du simplexe sur ce tableau pour trouver le maximum

si le maximum est 0 **alors**
renvoyer le tableau obtenu à la fin de 3., en réécrivant la fonction objectif de L avec les variables hors bases et en enlever x_0

//de toute façon on a $x_0 = 0$ donc on peut l'enlever
//l'espace des solutions est le même, mais il y a une solution basique. On peut remettre l'objectif
//Pour réécrire la fonction objectif, s'il y a une variable hors-base dans son expression on n'y touche pas. S'il y a une variable de base, on la remplace par son expression donnée par la contrainte correspondante

sinon
raise insatisfaisable

//l'espace des solutions est vide

Proposition 37 Le programme L_{aux} admet une solution.

Démonstration. Prendre $x = 0$ et $x_0 = -\min_i b_i$. ■

Proposition 38 L'espace des solutions de L est non vide ssi l'optimum de L_{aux} est 0.

Démonstration. $\boxed{\Rightarrow}$ Si l'espace est non vide, alors $Ax \leq b$ admet une solution. Mais alors on l'étend en prenant $x_0 = 0$. Et cela montre que l'optimum de L_{aux} est 0.
$\boxed{\Leftarrow}$ Réciproquement, si l'optimum de L_{aux} est 0, alors $Ax \leq b$. ■

Proposition 39 Après l'étape 2, le tableau admet une solution basique.

Démonstration. On note $x_N = (x_1, \ldots, x_d)$ les variables présentes dans L_{aux} sauf x_0. On note $x_{d+1}, \ldots, x_{d+m}$ les variables d'écart introduites.
Le tableau à l'étape 2 est :

$$\begin{cases} \text{maximiser } -x_0 \\ x_B = b + x_0 - Ax_N \\ x \geqslant 0 \end{cases}$$

La ligne du pivot est $x_{d+k} = b_{d+k} - (Ax_N)_{d+k} + x_0$. La nouvelle base contient les même variables sauf x_{d+k}, et avec x_0 en plus : $B' := B \cup \{0\} \setminus \{d+k\}$. Les variables hors bases (considérées comme nulles) sont x_N et x_k. La ligne du pivot devient : $x_0 = -b_k + (Ax_N)_k - x_{d+k}$. La valeur de x_0 est bien positive dans la solution basique. Pour les variable x_i avec i dans B', la ligne devient

$$x_b = b_b - b_k + (Ax_N)_k - x_{d+k}$$

Et donc la valeur de x_b dans la solution basique est $x_b \leq 0$.
On obtient donc :

$$\begin{cases} \text{maximiser } -x_k + b_k - (Ax_N)_k \\ x_0 = b_k - (Ax_N)_k \\ x_B = b - Ax_N + x_0 \\ x \geqslant 0 \end{cases}$$

■

Exemple 40 Le programme canonique suivant n'admet pas $(0,0)$ comme solution :

$$\begin{array}{l}\text{maximiser } 2x_1 - x_2 \\ \left\{\begin{array}{l} 2x_1 - x_2 \leq 2 \\ x_1 - 5x_2 \leq -4 \\ x_1, x_2 \geq 0 \end{array}\right.\end{array}$$

1. Programme auxiliaire.

$$\begin{array}{l}\text{maximiser } -x_0 \\ \left\{\begin{array}{l} 2x_1 - x_2 - x_0 \leq 2 \\ x_1 - 5x_2 - x_0 \leq -4 \\ x_1, x_2, x_0 \geq 0 \end{array}\right.\end{array}$$

2. Tableau du programme auxiliaire

$$\begin{array}{l}\text{maximiser } -x_0 \\ \left\{\begin{array}{l} x_3 = 2 - 2x_1 + x_2 + x_0 \\ x_4 = -4 - x_1 + 5x_2 + x_0 \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array}$$

3. Pivot. La variable de valeur la plus petite (-4) est x_4.

$$\begin{array}{l}\text{maximiser } -x_0 \\ \left\{\begin{array}{l} x_3 = 2 - 2x_1 + x_2 + x_0 \\ x_4 = -4 - x_1 + 5x_2 + x_0 \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array} \xrightarrow{x_4 := 0 \quad x_0 \nearrow} \begin{array}{l}\text{maximiser } -4 - x_1 + 5x_2 + x_4 \\ \left\{\begin{array}{l} x_3 = 6 - x_1 - 4x_2 + x_4 \\ x_0 = 4 + x_1 - 5x_2 - x_4 \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array}$$

4. Exécution de l'algorithme du simplexe. Le tableau courant admet une solution basique. On lance l'algo du simplexe pour voir si le max = 0. Si ce n'est pas le cas, on répond 'l'espace des solutions de L est vide'.

$$\begin{array}{l}\text{maximiser } -4 - x_1 + 5x_2 + x_4 \\ \left\{\begin{array}{l} x_3 = 6 - x_1 - 4x_2 + x_4 \\ x_0 = 4 + x_1 - 5x_2 - x_4 \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array} \xrightarrow{x_0 := 0 \quad x_3 \nearrow} \begin{array}{l}\text{maximiser } -x_0 \\ \left\{\begin{array}{l} x_3 = \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \\ x_2 = \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array}$$

L'algo du simplexe termine et le max vaut bien 0. L'espace des solution de L est non vide.

5. Réécriture de la fonction objectif. On remet l'objectif initial, que l'on réécrit avec les variables "nulles" (hors base), puis on enlève x_0 qui est nulle :

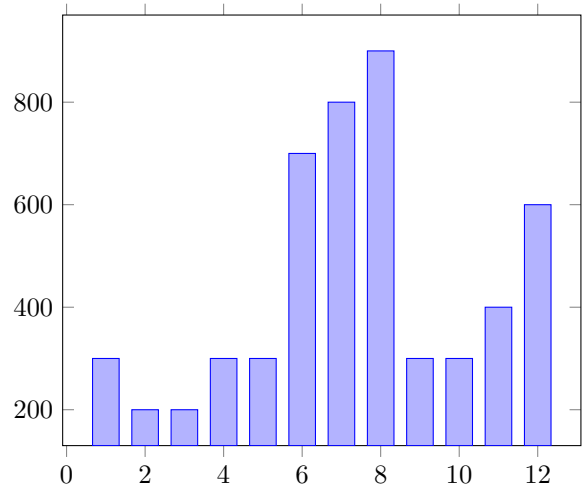
$$\begin{array}{l}\text{maximiser } 2x_1 - x_2 \\ \left\{\begin{array}{l} x_3 = \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \\ x_2 = \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array} \quad \begin{array}{l}\text{maximiser } -\frac{4}{5} + \frac{x_0}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ \left\{\begin{array}{l} x_3 = \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \\ x_2 = \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_1, x_2, x_0, x_3, x_4 \geq 0 \end{array}\right.\end{array}$$

$$\begin{array}{l}\text{maximiser } -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ \left\{\begin{array}{l} x_3 = \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \\ x_2 = \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_1, x_2, x_3, x_4 \geq 0 \end{array}\right.\end{array}$$

10 Exemples

10.1 Manger de la glace toute l'année

Nous sommes tous et toutes des gourmands et gourmandes. Nous voulons de la glace toute l'année. Nous gérons une entreprise de fabrication de glaces à la framboise (ou autre, ça n'a pas beaucoup d'importance). La demande $(d_i)_{i=1..12}$ en tonnes pour les 12 mois de l'année est donnée par ce graphique :



On suppose que la demande est connue (ou varie très peu d'une année à l'autre). C'est donc une entrée fiable du problème.

On veut minimiser le coût pour l'entreprise de la façon suivante.

— Premièrement, il faut conserver la glace en surplus d'un mois à l'autre. Ca coûte 20€ par tonne.

— Deuxièmement, il y a des changements administratifs chaque mois : recruter de nouveaux employé.e.s, ajuster les machines, virer des gens, etc. Les changements administratifs d'un mois à l'autre coûte 50€ par tonne de différence dans la production.

On veut :

— satisfaire la demande

— ne rien jeter

— avoir 0 surplus à la fin de l'année (pour des raisons légales ou je ne sais pas quoi)

Modélisation sous forme de programme (presque) linéaire .

On introduit les variables suivantes :

— x_i = production de glace au mois i

— s_i = surplus à la fin du mois i

minimiser $50 \sum_{i=2}^{12} |x_i - x_{i-1}| + 20 \sum_{i=1}^{12} s_i$

$$\begin{cases} x_i + s_{i-1} - d_i = s_i \\ s_0 = 0 \\ s_{12} = 0 \\ x_i, s_i \geq 0 \end{cases}$$

Se débarrasser de la valeur absolue. On introduit des variables (positives) y_i et z_i et on écrit $x_i - x_{i-1} = y_i - z_i$ que l'on rajoute comme contraintes. On remplace $|x_i - x_{i-1}|$ par $y_i + z_i$ dans l'objectif.

minimiser $50 \sum_{i=2}^{12} (y_i + z_i) + 20 \sum_{i=1}^{12} s_i$

$$\begin{cases} x_i + s_{i-1} - d_i = s_i \\ x_i - x_{i-1} = y_i - z_i \\ s_0 = 0 \\ s_{12} = 0 \\ x_i, s_i, y_i, z_i \geq 0 \end{cases}$$

Comme c'est un problème de minimisation, un des y_i ou z_i est nul.

10.2 Problème du flot de coût minimum

On considère un réseau de flots $G = (V, E, d, c, w, s, t)$ où :

— $d \in \mathbb{R}$ est un demande ;

— $c : E \to \mathbb{R}^+$ est la capacité (comme dans le problème du flot max)

— $w : E \to \mathbb{R}$ est le coût du transport d'une unité de flot pour un arc donné

— s sommet source,

— t somme puits

Le but est de savoir s'il existe un flot $f : E \to \mathbb{R}$ qui minimise le coût total $\sum_{e \in E} w(e) f(e)$ sous les contraintes de capacité $\forall e \in E, 0 \leq f(e) \leq c(e)$, de demande $\sum_{v \in V} f(s, v) = d$, et loi de Kirschoff $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$.

Remarque 41 On peut réduire à la programmation linéaire.

10.3 Problème du flot max de coût minimum

On cherche un flot max entre une source s et une destination t, qui est de coût min. Autrement dit, parmi les flot max, on veut celui de coût min. On calcule le flot max puis on demande une demande d qui faut le flot max.

10.4 Flot multi-produits

Même problème que le problème du flot de coût minimum mais avec k produits. Du coup, on a une source par produit, un puits par produit, et une demande par produit.

La réduction à la programmation linéaire est dans ce cas le meilleur algo connu. Attention, par contre, les solutions peuvent être rationnelles, même si les coeffs sont entiers.

11 FAQ

Pourquoi l'algorithme du simplexe s'appelle comme ça ? Parce qu'un simplexe est l'enveloppe convexe de points dont aucun est barycentre d'autres, en gros un polyèdre. Et l'algorithme se promène sur les sommets d'un tel polyèdre.

Pourquoi on parle de base dans variables de base ? Parce que A_B est inversible.

Comment gérer les inégalités strictes ? le problème d'optimisation n'est plus max/min mais sup/inf. On se souvient de quelles inégalités sont strictes et lesquelles sont larges. On résout avec inégalité large. Puis, si dans le tableau final, on est dans un sommet avec une inégalité stricte, on a un sup/inf. Sinon, s'il y a que des inégalité large dans le sommet optimal, c'est max/min.

Notes bibliographiques

L'exemple initial provient de [DPV08]. Ce livre vulgarise très bien la programmation linéaire. Malheureusement, il ne donne aucune démonstration.

Dans [CLRS09], un programme linéaire équationnel s'appelle un programme standard (mais l'adjectif standard n'a pas trop de sens). Ici, nous empruntons La terminologie de [GM07] : "programme sous forme équationnelle" que nous abrégeons en "programme équationnel". [CLRS09] donne des démonstrations mais parfois les justifications sont lourdes.

Comme mentionné dans [GM07], le plus petit exemple de programme linéaire où le simplexe boucle est donné dans Chvátal's textbook cited in Chapter 9 [CC+83].

Le prétraitement pour avoir un tableau avec solution basique vient de (cf. [CLRS09], p. 886, 29.5). J'ai fait un choix d'écrire explicitement les programmes linéaires et d'éviter d'écrire des appels cryptiques comme $pivot(N, B, A, b, c, v, \ell, 0)$, cf. [CLRS09] !

Les démonstrations dans [GM07] sont pas mal du tout, notamment la terminaison vient de là.

Références

[CC+83] Vasek Chvatal, Vaclav Chvatal, et al. *Linear programming.* Macmillan, 1983.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms.* McGraw-Hill, 2008.

[GM07] Bernd Gärtner and Jirí Matousek. *Understanding and using linear programming.* Universitext. Springer, 2007.

[Van98] Robert J. Vanderbei. *Linear programming - foundations and extensions*, volume 4 of *Kluwer international series in operations research and management service.* Kluwer, 1998.

ALGO1 – Algorithmes gloutons et matroïdes

François Schwarzentruber

Les algorithmes gloutons sont des algorithmes qui mangent le plus meilleur d'abord, un peu comme nous. La plupart du temps, les algorithmes gloutons ne sont pas optimaux, mais des fois oui. Dans ce cours, on regarde quelle est la structure algébrique nécessaire sur le problème algorithmique pour avoir un algorithme glouton optimal.

Il y a plusieurs structures algébriques qui ont été exhibé : matroïdes, greedoïdes, etc. Vous connaissez sans doute les structures algébriques de monoïdes, groupes etc. Ce n'est pas pareil. Ca ressemble plutôt à un espace topologique, ou à un espace mesurable, car on a un ensemble E et une collection de sous-ensembles $\mathcal{I}$. Un matroïde est $(E, \mathcal{I})$.

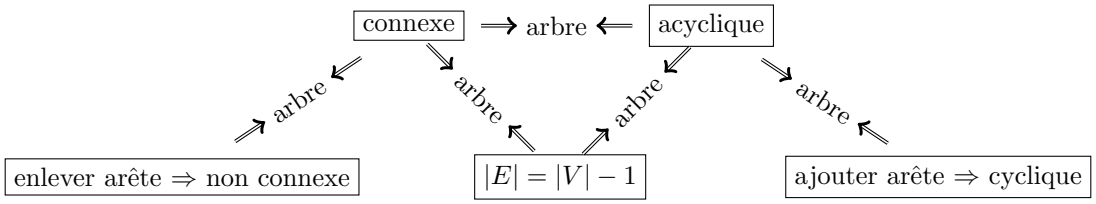
Mais avant de commencer dans le vif du sujet, soyons concret et parlons du problème de l'arbre couvrant minimal et de l'algorithme de Kruskal.

1 Caractérisation des arbres

Définition 1 (arbre) Un arbre est un graphe non orienté connexe et acyclique.

Proposition 2 (démonstration en exo) Soit $G = (V, E)$ un graphe non orienté. Sont équivalentes :

1) G est un arbre.
2) G est connexe et $|E| = |V| - 1$.
3) G est connexe mais, si l'on enlève une arête quelconque à E, le graphe ne l'est plus
3') Deux sommets quelconques de G sont reliés par une chaîne élémentaire unique.
4) G est acyclique et $|E| = |V| - 1$.
5) G est acyclique, mais si l'on ajoute une arête quelconque à E, il contient un cycle.

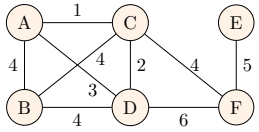


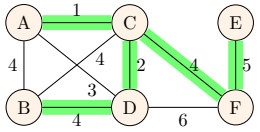
2 Arbre couvrant de poids minimal

Applications : connecter des sommets avec coût minimal.

	Sommets	Arêtes	Coût d'une arête
Circuits électroniques	Broches	Chemins où on peut faire couler de l'étain	Quantité d'étain
Pistes cyclables	Maisons	Routes	Longueur d'une route
Single-link clustering	Elements		Distance entre objets

Exemple 1





Définition 3 (arbre couvrant) Soit un graphe $G = (V, E, poids)$ non orienté pondéré avec $poids : E \to \mathbb{R}$. Un arbre couvrant de G est un sous-ensemble $T \subseteq E$ d'arêtes tel que (V, T) soit un arbre.

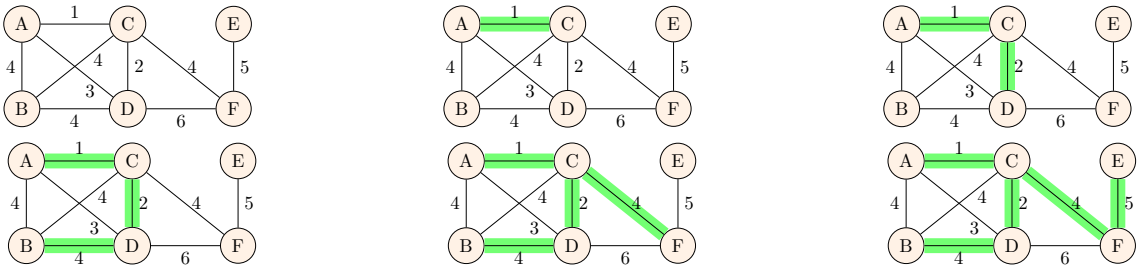
Définition 4 (arbre couvrant de poids minimal) Un arbre couvrant de poids minimum de G est un arbre couvrant avec $poids(T) = \sum_{(a,b)\in T} poids(a, b)$ est minimal.

Définition 5 Le problème de l'arbre couvrant de poids minimum est le problème suivant :
— entrée : un graphe non orienté **connexe** pondéré $G = (V, E, poids)$;
— sortie : un arbre couvrant $T \subseteq E$ de poids minimal.

3 Algorithme de Kruskal

```
pre : Un graphe G = (V, E, poids) pondéré non orienté connexe
post : Un arbre couvrant minimal de G
fonction kruskal(G)
    S := ∅
    trier les arêtes de E par poids croissant
    pour e ∈ E par poids croissant
        si e ne rajoute pas de cycle dans (V, S)
            ajouter e à S
    renvoyer S
```

Exemple 2



Théorème 6 kruskal(G) renvoie un arbre couvrant de poids minimal de G.

Démonstration. C'est une conséquence de la théorie des matroides. ■

4 Abstraction

Un peu d'abstraction dans ce monde de brut !

Définition 7 (système d'ensembles) Soit E un ensemble fini et $\mathcal{I}$ une collection de sous-ensembles de E. La donnée $(E, \mathcal{I})$ s'appelle un système d'ensembles.

Exemple 8 Dans l'exemple de l'algorithme de Kruskal, on prend :
— E est l'ensemble des arêtes dans G
— $\mathcal{I}$ est l'ensemble des forêts, i.e. des sous-ensemble d'arêtes sans cycles.

4.1 Problème d'optimisation générique

On se donne une fonction de poids $w : E \to \mathbb{R}$. Le poids d'un sous-ensemble $A \subseteq E$ est donnée par $\sum_{x \in A} w(x)$.
On peut s'intéresser à deux problèmes :

Définition 9 (Problème de minimisation abstrait)
— Entrée : un système d'ensembles $(E, \mathcal{I})$, une fonction de poids $w : E \to \mathbb{R}$;
— un sous-ensemble $A \in \mathcal{I}$ maximal (pour l'inclusion) de poids minimal.

Le problème de minimisation est un peu compliqué car il y a maximal-minimal. Considérons plutôt ce problème de maximisation :

Définition 10 (Problème de maximisation abstrait)
— Entrée : un système d'ensembles $(E, \mathcal{I})$, une fonction de poids $w : E \to \mathbb{R}$;
— un sous-ensemble $A \in \mathcal{I}$ de poids maximal.

Exemple 3 (Problème de la forêt couvrante de poids maximal)
— entrée : un graphe non orienté pondéré $G = (V, E, w)$
— sortie : une forêt $F \subseteq E$ avec $w(F)$ de poids maximal.

On réduit le problème de la forêt couvrante de poids maximal au problème abstrait de maximisation directement avec w. On prend $\mathcal{I}$ = ensembles des forêts dans E.

Exemple 4 (Problème de l'arbre couvrant de poids minimal)
On réduit le problème de l'arbre couvrant minimal au problème de la forêt couvrante de poids maximal comme suit (avec que des arêtes de poids strictement positifs d'ailleurs). On pose $w_0 = 1+$le poids de l'arête la plus lourde. On pose $w(e) = w_0 - \mathit{poids}(e)$. On prend $\mathcal{I}$ = ensembles des forêts dans E.

Une forêt w-maximale est un arbre poids-minimal. En effet, si jamais la forêt n'est pas connexe, alors on peut ajouter des arêtes et ça augmente le poids. Donc la forêt est bien un arbre et possède bien $|V| - 1$ arêtes. Puis $w(F) = (|V| - 1)w_0 - \mathit{poids}(F)$.

Dans les problèmes d'optimisation abstraits ci-dessus, l'ensemble E est donné explicitement, mais pas l'ensemble $\mathcal{I}$. L'ensemble $\mathcal{I}$ est généralement de taille exponentielle en $|E|$. Au lieu de ça, on se donne un oracle. Un oracle est un algorithme qui permet décider des choses. Ici, on se donne un oracle qui teste l'appartenance d'un ensemble A quelconque à $\mathcal{I}$.

4.2 Algorithme glouton générique

On donne ici un algorithme glouton pour le problème de maximisation.

```
fonction glouton(E, I, w)
    S := ∅
    trier les éléments de E par ordre décroissant de poids
    pour e ∈ E dans l'ordre décroissant de poids
        si S ∪ {e} ∈ I alors
            S := S ∪ {e}
    renvoyer S
```

Question 11 Quand l'algorithme glouton ci-dessus est-t-il correct ? La réponse que l'on va donner à cette question porte sur la structure de $(E, \mathcal{I})$.

Algorithme glouton pour le problème de minimisation

De même, on peut donner un algorithme glouton pour le problème de minimisation. C'est un peu le dual de l'algorithme qu'on va étudier nous.

```
fonction glouton(E, I, w)
    S := E
    trier les éléments de E par ordre croissant de poids
    pour e ∈ E dans l'ordre croissant de poids
        si S \ {e} inclut une base alors
            S := S \ {e}
    renvoyer S
```

5 Théorie des Matroïdes

Un matroïde (et en version plus faible un système d'indépendance) a comme but de modéliser la notion d'indépendance au sens abstrait. Un exemple concret correspondant à cette notion vient des matrices. Considérons une matrice M de taille $n \times n$ (c'est pareil que de considérer n vecteurs de $\mathbb{R}^n$). Un sous-ensemble A de lignes $\{M_{i_1}, \ldots, M_{i_k}\}$ de M est indépendants si $\sum_{j=1}^{k} \lambda_j M_{i_j} = (0, \ldots, 0)$ implique $\lambda_1 = \cdots = \lambda_k = 0$.

5.1 Système d'indépendance

On modélise la notion d'indépendance de manière abstraite avec un système d'ensembles.

Exemple 12 E = ensemble des lignes d'une matrice A. $\mathcal{I}$ = collection des sous-ensembles de lignes qui sont indépendantes.

De manière générale, un tel système vérifie les propriétés données dans la définitino suivante, et on dira alors que c'est un système d'indépendance.

Définition 13 (système d'indépendance) Un système d'ensemble $(E, \mathcal{I})$ est un système d'indépendance si :

1. $\emptyset \in \mathcal{I}$;

2. (clos par sous-ensemble) $A \in \mathcal{I}$ et $B \subseteq A$ implique $B \in \mathcal{I}$;

Malheureusement, un système d'indépendance ne suffit pas pour avoir un algorithme correct (on le verra plus tard). Mais cela ne nous empêche pas de modéliser quelques problèmes algorithmiques, car l'algorithme, bien que non correct, sera (au pire !) un algorithme d'approximation.

Exemple 14 Trouver un ensemble indépendant dans un graphe de poids maximum
E = ensemble des sommets du graphe
$\mathcal{I}$ = sous-ensemble de sommets indépendants

Exemple 15 (voyageur de comerce) E = ensemble des arêtes du graphe
$\mathcal{I}$ = sous-ensemble des arêtes d'un cycle hamiltonien

Exemple 16 (plus court chemin de s à t) E = ensemble des arêtes du graphe
$\mathcal{I}$ = sous-ensemble des arcs d'un chemin de s à t

⚠ Attention, pour cet exemple, le test $A \in \mathcal{I}$ est coûteux (décider si un sous-ensemble d'arcs est inclus dans un chemin est NP-complet je crois).

Exemple 17 (problème du sac à dos) E = ensemble $\{1, \ldots, n\}$ des objets
$\mathcal{I}$ = sous-ensemble d'objets qui ne dépassent pas le poids seuil

Exemple 18 (problème du couplage de poids maximum dans un graphe non orienté) E = ensemble des arêtes
$\mathcal{I}$ = ensemble des couplages de G

5.2 Matroïdes

Mais avec la notion de matroïdes ça va mieux... et ça tombe bien, l'exemple des lignes d'une matrice fonctionne toujours. C'est en fait un matroïde. Un matroïde vérifie en plus la propriété d'échange que voici. C'est en lien avec le théorème de complétion des bases aussi appelé *théorème de la base incomplète* en algèbre linéaire.

Définition 19 (matroïde) Un système d'indépendance $(E, \mathcal{I})$ est un matroïde si de plus on a :

3. (échange) $A, B \in \mathcal{I}$ et $|A| < |B|$ implique il existe $e \in B \setminus A$ tel que $A \cup \{e\} \in \mathcal{I}$.

Cette propriété d'échange est vraie pour les forêts, autrement dit, ce que Kruskal a besoin. On définit maintenant un cas particulier de matroïdes qui est basé sur les graphes non orientés et qui coincident exactement avec ce qu'il nous faut pour l'analyse de l'algorithme de Kruskal.

Proposition 20 Soit $G = (V, E)$ un graphe non orienté. Alors $M_G = (E, \mathcal{I})$ où $\mathcal{I}$ est l'ensemble des forêts est un matroïde.

Démonstration. Le point à vérifier, c'est la propriété d'échange. Elle dit : soit $F, F' \in \mathcal{I}$, si $|F| < |F'|$, alors il existe $e \in F' \setminus F$, tel que $F \cup \{e\} \in \mathcal{I}$. Montrons la contraposée.
Supposons que pour tout $e \in F' \setminus F$, on a $F \cup \{e\} \notin \mathcal{I}$. Montrons que $|F'| \leq |F|$.

Notation 21 Notons F^* la relation 'être dans un même arbre dans F' (c'est une étoile comme fermeture de Kleene, fermeture réflexive et transitive de F vu comme une relation). Écrire xF^*y c'est dire qu'il y a chemin de x à y composé d'arêtes de F. De même pour F'^*.

Fait 22 $F' \subseteq F^*$.

Démonstration. Soit $uF'v$. Montrons que uF^*v. Deux choix : soit uFv et c'est bon. Ou alors d'après l'hypothèse, on a $F \cup \{(u, v)\} \notin \mathcal{I}$. Mais $F \in \mathcal{I}$. Donc ça veut dire que l'ajout de (u, v) à F crée un cycle. Il y avait donc déjà un chemin de u à v dans F. Autrement dit uF^*v. ■
On appelle $\#F$-composante une forêt dans F (c'est aussi une classe d'équivalence pour la relation F).

Fait 23 $\#F$-composantes $\leq \#F'$-composantes.

Démonstration. Comme $F' \subseteq F^*$, on a $F'^* \subseteq F^*$. On a donc $\#F$-composantes $\leq \#F'$-composantes. ■

Lemme 24 Une forêt $F \subseteq E$ contient $|V| - |F|$ composantes.

Démonstration.

$$\begin{aligned}|F| &= \sum_{i=1}^{\#\text{ arbres}} \#\text{ arêtes dans l'arbre numéro } i \\ &= \sum_{i=1}^{\#\text{ arbres}} \#\text{ sommets dans l'arbre numéro } i-1 \\ &= |V| - \#\text{ arbres}\end{aligned}$$

■

De $\#F$-composantes $\leq \#F'$-composantes, on obtient $|V| - |F| \leq |V| - |F'|$. Donc $|F'| \leq |F|$.

■

Définition 25 (matroïde graphique) Le matroïde M_G s'appelle le matroïde graphique.

5.3 Bases

Définition 26 (ensemble indépendant maximal) Un ensemble indépendant $A \in \mathcal{I}$ est maximal si pour tout $e \in E \setminus A$, $A \cup \{e\} \notin \mathcal{I}$.

Utilisons la terminologie d'algèbre linéaire :

Définition 27 (base) Une base est un ensemble indépendant maximal pour l'inclusion.

On énonce maintenant le fait que toutes les bases ont le même cardinal.

Proposition 28 Tous les bases ont le même cardinal.

Ca marche aussi si on se restreint à des ensembles indépendants inclus dans un sous-ensemble F de E. Ces ensembles s'appellent des *bases* de F.

Définition 29 Un ensemble indépendant A inclus dans F et maximal pour l'inclusion s'appelle base de F : pour tout $e \in F \setminus A$, $A \cup \{e\} \notin \mathcal{I}$.

Proposition 30 Soit $F \subseteq E$. Tous les bases de F ont le même cardinal.

Démonstration. Par la propriété d'échange ! Par l'absurde, prenons deux sous-ensembles indépendants maximaux Y et Y' avec, sans perte de généralité $|Y| < |Y'|$. La propriété d'échange dit qu'il existe $e \in Y' \setminus Y$ avec $Y \sqcup \{e\}$ indépendant. Du coup, Y n'est pas maximal. ■

6 Correction de l'algorithme glouton

On suppose que $(E, \mathcal{I})$ est un ensemble d'indépendance. On va montrer que l'algorithme abstrait sur $(E, \mathcal{I})$ renvoie une solution optimale ssi $(E, \mathcal{I})$ est un matroïde.

Mieux que ça, on a un algorithme d'approximation, en lien avec plusieurs notions de rangs.

6.1 Rangs

Définition 31 Le rang de $F \subseteq E$ noté $r(F)$ est le cardinal du plus grand ensemble indépendant inclus dans F :

$$r(F) = \max\{|Y| : Y \subseteq F, Y \in \mathcal{I}\}.$$

Définition 32 Le rang inférieur $\rho(F)$ est le cardinal du plus petit ensemble indépendant que l'on ne peut pas agrandir en y ajoutant un élément. Formellement :

$$\rho(F) := \min\{|Y| : Y \subseteq F, Y \in \mathcal{I} \text{ et pour tout } x \in F \setminus Y,\ Y \sqcup \{x\} \notin \mathcal{I}\}.$$

Proposition 33 $\rho(F) \leq r(F)$.

Démonstration. Soit Y tel que $|Y| = r(F)$. Alors Y est maximal dans F : pour tout $x \in F \setminus Y$, $Y \cup \{x\} \notin \mathcal{I}$ (car sinon, $r(F) \geq |Y \sqcup \{x\}) > |Y|$). Autrement dit Y fait partie des sous-ensembles balayés dans la définition du min dans $\rho(F)$. Ainsi, $\rho(F) \leq |Y|$. ■

Proposition 34 Si $(E, \mathcal{I})$ est un matroide, alors $\rho(F) = r(F)$.

DÉMONSTRATION. Soit Y tl que $|Y| = r(F)$. Alors Y est maximal dans F. $\rho(F)$ est le min de $|Y|$ sur tous les Y maximaux inclus dans F... mais ils ont tous même cardinal. Donc $\rho(F) = r(F)$. ■

Définition 35 (rang quotient)

$$q(E, \mathcal{I}) := \min_{F \subseteq E} \frac{\rho(F)}{r(F)}.$$

Proposition 36 $q(E, \mathcal{I}) \leq 1$.

Proposition 37 $(E, \mathcal{I})$ est un matroïde ssi $q(E, \mathcal{I}) = 1$.

DÉMONSTRATION. $\boxed{\Rightarrow}$ Supposons que $(E, \mathcal{I})$ est un matroïde. On a vu que $\rho(F) = r(F)$ pour tout F. Donc $q(E, \mathcal{I}) = 1$.

$\boxed{\Leftarrow}$ Supposons $q(E, \mathcal{I}) = 1$. Pour tout F, $\rho(F) = r(F)$. Cela signifie que toutes les bases de F ont même cardinal. On montre que ça implique la propriété d'échange. Soit A et B indépendants avec $|A| < |B|$. Ainsi, comme toutes les bases $F = A \cup B$ ont le même cardinal, A n'est pas une base de $A \cup B$. Il existe donc $e \in \underbrace{A \cup B \setminus A}_{B \setminus A}$ avec $A \sqcup \{e\}$ indépendant. ■

Théorème 38 (Jenkyns en 1976, Korte et Hausmann 1978) Soit $(E, \mathcal{I})$ est un système d'indépendance. Soit w une fonction de poids. Soit S une solution de l'algorithme. Soit S^* une solution optimale.

Si $(E, \mathcal{I})$ est un système d'indépendance alors

$$q(E, \mathcal{I}) \leq \frac{w(S)}{w(S^*)} \leq 1.$$

De plus, on peut trouver une fonction de poids où la borne inférieure est atteinte.

DÉMONSTRATION. Soit S une solution de l'algorithme. Soit S^* une solution optimale.

Soit $E = \{e_1, \ldots, e_n\}$ où $e_1, \ldots, e_n$ est l'ordre des éléments considérés par l'algorithme, i.e. par poids décroissant :

$$w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n).$$

On note pour tout $j \in \{'\}0, 1, \ldots, n$, $E_j = \{e_1, e_2, \ldots, e_j\}$, $S_j = S \cap E_j$, $S_j^* = S^* \cap E_j$ (pour $j = 0$, ces ensembles sont vides). On a :

$$\begin{aligned}
w(S) &= \sum_{j \in S} w(e_j) \\
&= \sum_{j=1}^{n} 1_{j \in S} w_{e_j} \\
&= \sum_{j=1}^{n} (|S_j| - |S_{j-1}|) w(e_j) \\
&= \sum_{j=1}^{n} |S_j| (w(e_j) - w(e_{j+1})) \\
&\geq \sum_{j=1}^{n} \rho(E_j) (w(e_j) - w(e_{j+1})) \\
&\geq q(E, \mathcal{I}) \sum_{j=1}^{n} r(E_j) (w(e_j) - w(e_{j+1}) \\
&\geq q(E, \mathcal{I}) \sum_{j=1}^{n} |S_j^*| (w(e_j) - w(e_{j+1}) \\
&= q(E, \mathcal{I}) \sum_{j=1}^{n} (|S_j^*| - |S_{j-1}^*|) w(e_j) \\
&= q(E, \mathcal{I}) w(S^*)
\end{aligned}$$

Explication des calculs

Fait 39 $|S_j| \geq \rho(E_j)$.

Démonstration. On rappelle que $\rho(E_j)$ est le min sur $|Y|$ avec des $Y \subseteq E_j$, Y indépendant et Y maximal pour l'inclusion. Or S_j est un tel Y :

- S_j est inclus dans E_j
- S_j est indépendant par construction de l'algorithme
- Montrons que S_j est maximal pour l'inclusion. Par l'absurde. Supposons que l'on puisse ajouter un $e_k \in E_k \setminus S_j$ tout en restant indépendant. Autrement dit, $S_j \sqcup \{e_k\}$ indépendant. Mais alors $S_{k-1} \sqcup \{e_k\}$ est indépendant. Mais alors e_k aurait du être ajouté par l'algorithme et donc $e_k \in S_k \subseteq S_j$. Contradiction.

■

Fait 40 $\rho(E_j) \geq q(E, \mathcal{I}) r(E_j)$.

Démonstration. car $q(E, \mathcal{I})$ est le min des $\frac{\rho(F)}{r(F)}$. ■

Fait 41 $r(E_j) \geq |S_j^*|$.

Démonstration. car $r(E_j)$ est le max sur les $|Y|$ où $Y \subseteq E_j$ et Y indépendant. S_j^* est un tel Y. ■

Egalité pour une fonction de poids spécifique. Soit $F \subseteq E$ qui atteint le minimum du quotient $\frac{\rho(F)}{r(F)}$. Soit B_1, B_2 des bases de F, c'est-à-dire des ensembles indépendants maximaux de $\mathcal{I}$, tel que $|B_1| = r(F)$ et $|B_2| = \rho(F)$.

L'idée est que B_1 est maximal dans le sens où on ne peut plus rien ajouter, mais B_2 est peut-être plus grand.

Du coup, on va faire que B_1 soit choisi par l'algorithme en mettant les éléments de B_1 en premier. On suppose que l'ordre est $e_1 \geq \cdots \geq e_n$ avec $B_1 = \{e_1, \ldots, e_{|B_1|}\}$. On fait $w = 1_F$. Ainsi, l'optimum est de choisir un ensemble maximal dans F, donc de poids (et cardinal) $|B_2|$. Alors que l'algorithme, lui, va choisir B_1. ■

Corollaire 42 L'algorithme abstrait est optimal sur $E, \mathcal{I}$ pour tout poids ssi $(E, \mathcal{I})$ est un matroïde.

En algorithmique, je pense que c'est la seule caractérisation algèbrique que vous allez voir de votre vie.

Exercice 1 Calculer la borne d'approximation pour les exemples plus haut. Donner des exemples sympas.

7 Ordonnancement

Idée de TD

Définition 43 (ordonnancement de tâches de durée 1 sur un unique processeur)

- un ensemble de n tâches, où la tâche numéro i est décrite par une deadline d_i et une pénalité w_i ;
- une permutation qui minimise la pénalité globale (on compte la pénalité w_i si la tâche i est faite après d_i), ou qui maximise les pénalités non attribués (i.e. la somme des pénalités des tâches tôts, qui finissent avant leurs deadlines)

Définition 44 (tâche tôt, tâche tard) Une tâche est tôt quand elle est exécuté avant sa deadline. Elle est tard sinon.

Définition 45 (ordonnancement sous forme canonique) Un ordonnancement est sous forme canonique si on a d'abord les tâches tôts puis les tâches tards, et que les tâches tôts sont triés par ordre de deadlines croissantes.

Proposition 46 Toute permutation est équivalente à avoir les tâches tôts puis les tâches tards.

Démonstration. On échange une tâche tard avant une tâche tôt. On échange deux tâches tôts consécutives dans le mauvais ordre. ■

Définition 47 (ensemble indépendant) Un ensemble de tâches A est indépendant s'il existe un ordonnancement dans lequel les tâches de A sont tôts.

Définition 48 On définit $(E, \mathcal{I})$ avec E un ensemble de tâches et $\mathcal{I}$ l'ensemble des ensembles indépendants de tâches.

Théorème 49 $(E, \mathcal{I})$ est un matroïde.

Corollaire 50 On peut utiliser l'algorithme générique.

Apprentissage supervisé

François Schwarzentruber

Pourquoi apprendre ? Pour ne pas juste écrire un algorithme qui résout un problème ? Il y a des situations où on ne sait pas comment programmer une tâche, même le ou la meilleur ou meilleure des programmeurs programmeuses. Par exemple pour le problème suivant :

- entrée : une image bitmap ;
- sortie : oui, s'il y a un chat dans l'image, non sinon.

C'est assez bizarre car un enfant peut reconnaître lui un chat en en ayant vu 2 ou 3. Bon, il est un peu bizarre de dire qu'un algorithme qu'il est intelligent alors qu'il a besoin d'une pléthore d'images pour reconnaître un chat. Mais soit. C'est ce type de problème que nous allons étudier dans ce cours, mais de manière assez générale.

Parfois, on saurait programmer. Mais c'est trop couteux de tout anticiper. Par exemple, on peut essayer d'anticiper tous les mouvements à faire pour les jambes d'un robot pour qu'il puisse marcher sur toutes les surfaces. Mais ce n'est juste pas raisonnable à construire.

1 Généralités sur l'apprentissage

Il existe trois grandes familles de techniques d'apprentissage.

1. L'apprentissage non-supervisé prend des données et cherche à les regrouper tout seul, sans qu'on l'aide, sans qu'on lui donne d'étiquettes. Par exemple, avec le temps, une voiture autonome peut développer le concept de 'condition de trajet fluide' et de 'bouchons'.
2. L'apprentissage par renforcement permet d'apprendre à bien jouer/prendre des actions dans des expériences qui s'écoulent dans le temps. Des actions peuvent avoir un impact dans le futur. L'agent reçoit des récompenses (ou punitions) en fonction de ce qu'il fait.
3. L'apprentissage supervisé (que l'on étudie dans ce cours), lui, consiste à partir de données étiquetés. Par exemple : des images de chats et des images qui ne contiennent pas de chats. A partir de ces données, il s'agit par exemple de construire un système qui reconnaît les chats.

Bon, en réalité, il y a un continuum entre l'apprentissage non-supervisé et supervisé. En effet, les données peuvent être mal étiquetés (exemple : des gens mentent sur leur âge), ou alors certaines données ne sont pas étiquetés etc.

2 Généralités sur l'apprentissage supervisé

Comment construire un algorithme qui réponde au problème suivant :

- entrée : une image de taille 64×64 ;
- sortie : dire si l'image contient un chat, un chien ou si elle représente autre chose.

2.1 Prédiction

Soit $\mathcal{X}$ un ensemble d'observations. Typiquement $\mathcal{X} = \mathbb{R}^d$ ou ici l'ensemble des images de taille 64×64. On se donne un ensemble d'étiquettes $\mathcal{Y}$. Par exemple $\mathcal{Y} = \{chat, chien, autrechose\}$.

Le but ultime est de construire un système capable de prédire l'étiquette d'une certaine observation :

- entrée : une observation x dans $\mathcal{X}$;

- sortie : l'étiquette dans $\mathcal{Y}$ prédite de x.

Dit autrement, il s'agit de construire une fonction $h : \mathcal{X} \to \mathcal{Y}$ qui prédit correctement.

Définition 1 (hypothèse / modèle) La fonction h s'appelle une *hypothèse* ou un *modèle*.

2.2 Jeu de données / dataset

Pour construire un tel système, on s'appuie sur l'apprentissage à partir d'un jeu de données / data set.

Définition 2 (dataset) Un dataset est un ensemble d'exemples observation/étiquette, i.e. $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$ où les $x_1, \ldots, x_n \in \mathcal{X}$ et $y_1, \ldots, y_n \in \mathcal{Y}$.

Exemple 3 Voici un exemple où les x_i sont des images bitmaps (matrices de taille 32×32) :

$\mathcal{D} = \{($  $, chat), ($  $, chien), ($  $, chien), ($  $, bouteille)\}$

Une observation est une image et son étiquette a été ajouté à la main (ou alors par un autre programme) et indique la classe de l'image (un chat, un chien, etc.).

Exemple 4 Voici un exemple où les x_i sont des points dans le plan $\mathbb{R}^2$:
$\mathcal{D} = \{((1,2), ok), ((1,3), \text{raté}), ((0,-5), ok), \ldots\}$

2.3 Problème d'apprentissage supervisé

Définition 5 (jeu d'apprentissage) Le jeu de données utilisé pour l'apprentissage s'appelle le jeu d'apprentissage.

Voici le problème générique d'apprentissage supervisé. Il s'agit de construire automatiquement un modèle $h : \mathcal{X} \to \mathcal{Y}$ satisfaisante, à partir d'un jeu d'apprentissage. Bon, le modèle n'est pas une fonction quelconque. On parle généralement de classes de modèles (aussi appelé espace des hypothèses), que l'on note $\mathcal{H}$.

Exemple 6
- $\mathcal{H} :=$ {fonctions représentables par un réseau de neurones avec 10 neurones et 6 couches}
- $\mathcal{H} :=$ {fonctions représentables par un arbre de décision}
- $\mathcal{H} :=$ {polynômes de degré 10}
- $\mathcal{H} :=$ {fonctions linéaires}
- $\mathcal{H} :=$ {fonctions définissant l'étiquette comme un vote majoritaire de celles des k plus proches voisins, où $k \geq 1$}

Définition 7 (problème d'apprentissage supervisé)

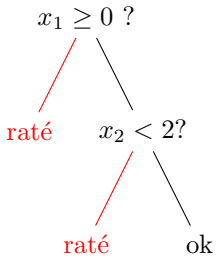
- entrée : un jeu d'apprentissage $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$;
- sortie : un modèle $h : \mathcal{X} \to \mathcal{Y}$ dans $\mathcal{H}$ qui colle au mieux au jeu d'apprentissage et qui prédit bien sur d'autres données.

Définition 8 (hypothèse consistante) Un modèle h est consistant avec $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$ si $h(x_i) = y_i$ pour tout $i = 1..n$.

Définition 9 (classification) Quand $\mathcal{Y}$ est fini, on parle de classification.

Définition 10 (régression) Quand $\mathcal{Y} = \mathbb{R}$, on parle de régression.

Exemple 11 La fonction $h : \begin{array}{ccc} \mathbb{R}^2 & \to & \{0,1\} \\ (x_1, x_2) & \mapsto & \begin{cases} \text{ok si } x_1 \geq 0 \text{ et } x_2 < 2 \\ \text{raté sinon.} \end{cases} \end{array}$ est représentée par l'arbre de décision



Exemple 12 Quid de $\mathcal{H}$ = ensemble des fonctions représentables par un programme en Rust ?

On aura pas de bons algorithmes d'apprentissage. On risque d'apprendre un programme compliqué. Un autre problème est que l'on veut que le calcul de $h(x)$ soit rapide. Avec un arbre de décision, un modèle linéaire, un réseau de neurones, la rapidité du calcul de $h(x)$ est garantie.

2.4 A-t-on bien appris ?

Le but est d'apprendre une hypothèse mais qui est bonne au delà du jeu d'apprentissage.

Définition 13 (généralisation) On dit que l'hypothèse se généralise bien si elle prédit correctement la valeur de y sur de nouveaux exemples.

Définition 14 (rasoir d'Ockham) Le principe du rasoir d'Ockham consiste à apprendre une hypothèse la plus simple possible.

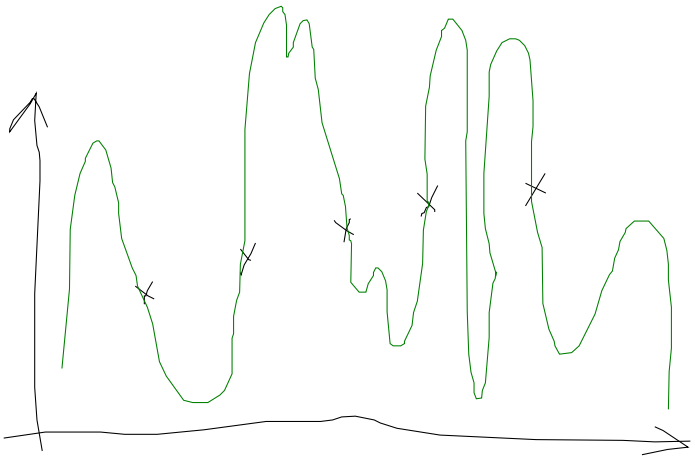
TODO: est-ce que modèle et hypothèse c'est la même chose ?

On veut généralement une hypothèse simple car elle va mieux se généraliser : elle va pouvoir mieux prédire sur de nouveaux exemples.

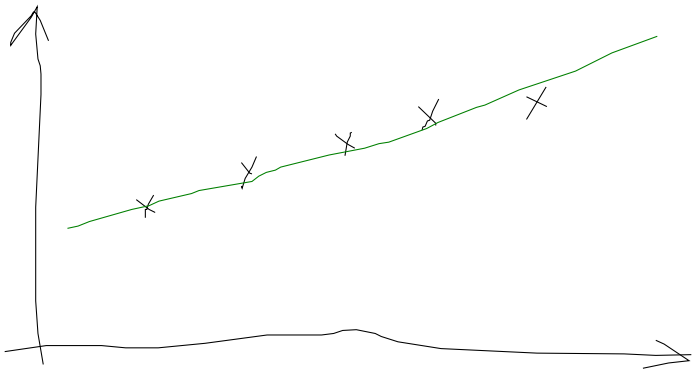
Définition 15 (overfitting) L'overfitting (aussi appelé sur-apprentissage) est le phénomène quand le modèle colle trop aux données d'apprentissage.

L'overffiting apparaît si l'espace des hypothèses est trop grand.

Exemple 16 Par exemple avec $\mathcal{H}$ = ensemble des fonctions polynomiales, on risque d'avoir de l'overfitting. Sur les données, l'hypothèse h est super, mais ailleurs c'est vraiment le bazar :



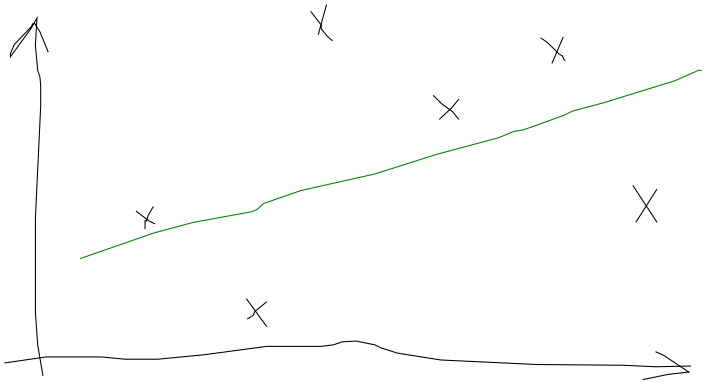
Un modèle linéaire se généralise mieux et on a a priori pas d'overffiting. D'une part, ça colle aux données mais en plus il y a de grandes chances que la réalité soit proche d'un modèle linéaire. Du coup on prédit bien :



L'overfitting disparait quand on prend plus d'exemples.

Définition 17 (underfitting) L'underfitting (aussi appelé sous-apprentissage) est le phénomène quand le modèle est trop simple pour comprendre les données.

Exemple 18 Voici un exemple de couples $(x_i, y_i) \in \mathbb{R}^2$ que l'on essaie d'approximer avec une fonction linéaire. C'est une situation d'underfitting car on ne capture pas bien les données.



Pour mesurer la qualité de notre hypothèse, on se base sur un jeu de test.

2.5 Jeu de test

Définition 19 (jeu de test) Le jeu de test est le jeu de données utilisé pour mesurer la qualité de notre hypothèse.

La matrice de confusion permet d'avoir une idée si l'algorithme d'apprentissage a bien fonctionné. Par exemple, pour chaque chat (classe réelle), on indique le nombre de chats classés 'chat', le nombre de chats classés 'fourchette'.

Définition 20 (matrice de confusion) Etant donné un dataset $\mathcal{D}$, et une hypothèse $h : \mathcal{X} \to \mathcal{Y}$, la matrice de confusion associée à $\mathcal{D}$ et f est la matrice $C = (C_{ce})_{c,e \in \mathcal{Y}}$ avec $C_{ce} = |\{i = 1..n \mid y_i = c \text{ and } h(x_i) = e\}|$.

Exemple 21 Voici un exemple :

	'chat'	'fourchette'
chat	95	5
fourchette	3	97

Parfois les catégories sont binaires.

Exemple 22 Voici un exemple :

	'courrier'		'pourriel'	
courrier	95 vrais positifs	Bonjour, c'est Bob. J'espère que tu vas bien. Tu peux venir à 17h. Bonne journée.	5 faux négatifs	Bonjour, c'est Bob. Ta blague de sexe m'a fait rire. A bientôt. Bonne journée.
pourriel	3 faux positifs	Bonjour, nous n'avons pas reçu votre attestation. Veuillez nous contacter.	97 vrais négatifs	Bonjour, voici une photo avec de la nudité qui peut vous intéresser.

3 Méthode du plus proche voisin

☺ simple à mettre en oeuvre
☺ très expressif (ça s'adapte complétement aux données !)
☹ coûteux en mémoire (le modèle lui-même contient le jeu de données !)
☹ besoin de définir une distance
☹ sensible au bruit (i.e. à une observation mal étiquetée) ← traité par la méthode des k plus proches voisins
☹ plus on a de données, plus la prédiction est lente

3.1 Définition

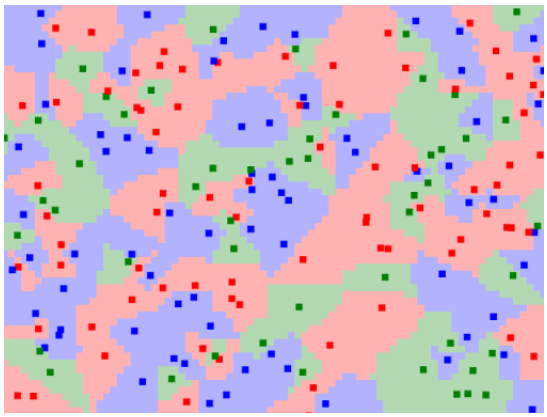
On suppose que l'on dispose d'une distance δ sur $\mathcal{X}$. Étant donné un dataset $\mathcal{D}$, on prédit d'étiquetter x par le label du x_i le plus proche de x selon la distance δ.

Définition 23 (apprentissage supervisé par méthode du plus proche voisin)

- entrée : un jeu d'apprentissage $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$;
- sortie : la fonction $\begin{array}{rccl} h: & \mathcal{X} & \to & \mathcal{Y} \\ & x & \mapsto & y_i \text{ où } i = argmin_{i\in\{1,\dots,n\}}\delta(x, x_i) \end{array}$

On peut reformuler h par l'algorithme suivant :

fonction $h(x)$
  calculer $i \in \{1, \dots, n\}$ tel que $\delta(x, x_i)$ minimal
  renvoyer y_i



En fait, c'est comme si on calculait le diagramme de Voronoi. Puis on affecte à x la classe du centroïde x_i dans lequel x se loge.

3.2 Implémentation

Naïvement, le calcul du plus proche voisin est en $O(n)$ calcul de distances à chaque fois. Si par exemple, on a $\mathcal{X} = \mathbb{R}^d$, alors le calcul d'une distance entre deux points est en $O(d)$. Ainsi :

Proposition 24 On peut calculer le plus proche voisin en $O(nd)$.

Mais on peut réaliser un précalcul sur les points $x_1, \dots, x_n$ pour que la recherche du plus proche voisin soit plus efficace.

3.2.1 Précalcul

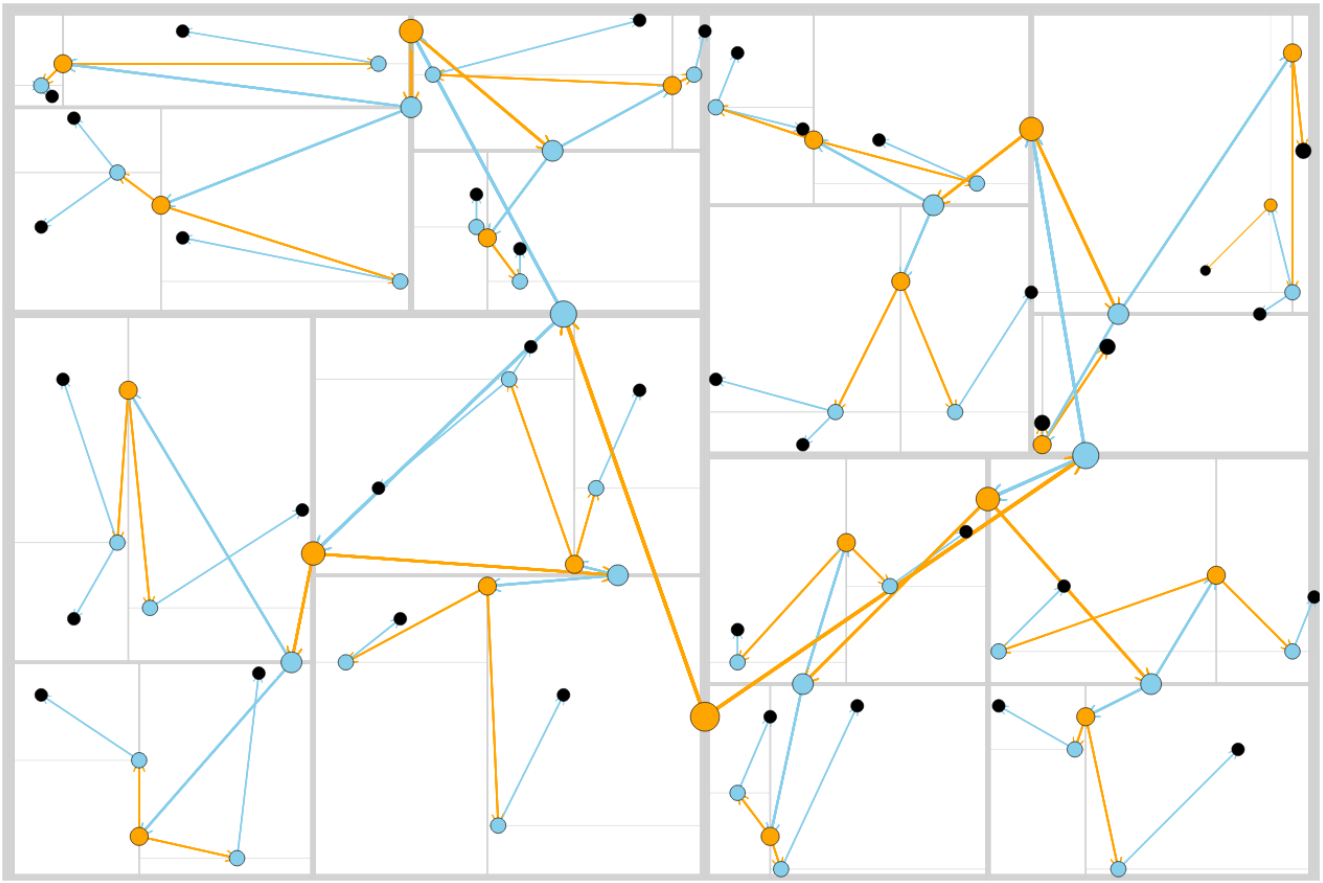
Arbre binaire de recherche

Un arbre binaire de recherche est un arbre d'une dichotomie sur des données a priori rangé selon une seule dimension. Les kd-arbres généralisent les arbres binaires de recherche à des points dans un espace à dimension quelconque.

On considère $\mathbb{R}^d$ où les dimensions sont numérotées $0, 1, \ldots, d-1$.

Définition 25 (kd-arbre) Un kd-arbre est un arbre binaire où chaque nœud est étiqueté par un point dans $\mathbb{R}^d$ où en chaque nœud interne de profondeur i le fils gauche ne contient que des points dont la coordonnée $i \mod d$ est inférieur ou égale, le fils droit ne contient que des points dont la coordonnée $i \mod d$ est supérieure.

Mais un prétraitement peut stocker nos points $x_1, \ldots, x_n$ dans une structure arborescente appelée k-d-arbres. Afin que le kd-arbre soit assez équilibré, on construit en calculant le médian des points selon la coordonnée adéquate. La racine sera le point médian selon la 1ère coordonnée. Dans le fils gauche, on stocke les points de 1ère coordonnée inférieure, à droite, les points de 1ère coordonnée supérieure. La racine du fils gauche est le point médian selon la 2e coordonnée etc. Plus précisément, si les coordonnées sont de 0, à $d-1$, à la profondeur i, on branche sur la coordonnée numéro $i \mod d$. Voici un exemple de kd-arbre en dimension $d = 2$:



3.2.2 Construction

Notations. Soit T un sous-arbre non vide d'un *kd*-arbre. On note

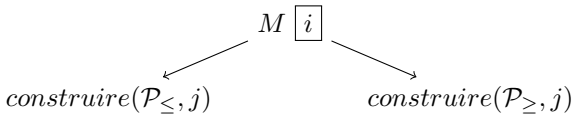
$$T = \begin{array}{ccc} & P\ \boxed{i} & \\ \swarrow & & \searrow \\ G & & D \end{array}$$

où P est le point sur lequel on branche en T, G est le sous-arbre gauche et D le sous-arbre droit, i est la dimension sur laquelle on branche. On note $\emptyset$ l'arbre vide.

L'hyperplan de T est $\{P' \in \mathbb{R}^d \mid P'[i] = P[i]\}$.

entrée : un ensemble de points $\mathcal{P}$
sortie : un *kd*-arbre qui contient exactement les points de $\mathcal{P}$
fonction construire($\mathcal{P}$, $i = 1$)
  si $\mathcal{P} = \emptyset$ **alors**
    renvoyer $\emptyset$
  sinon
    écrire $\mathcal{P} := \mathcal{P}_{\leq} \cup \{M\} \cup \mathcal{P}_{\geq}$
    où
    - $|\mathcal{P}_{\leq}| - |\mathcal{P}_{\geq}| = 0$ ou 1 et
    - pour tout $P \in \mathcal{P}_{\leq}$, $P[i] \leq M[i]$
    - pour tout $P \in \mathcal{P}_{\geq}$, $P[i] \geq M[i]$

    renvoyer



Proposition 26 La construction d'un kd-arbre à n points est en temps $\Theta(n \log n)$ et en espace $\Theta(n)$.

Démonstration. On peut calculer un médian en temps $\Theta(n)$ et en espace $\Theta(n)$ (vous savez comment ?). Soit $T(n)$ la complexité de construction d'un kd-arbre. On a : $T(1) = \Theta(1)$ et $T(n) = 2T(n/2) + \Theta(n)$ pour $n > 1$. D'où $T(n) = \Theta(n \log n)$.

Pour l'espace, à la fin, il y a un nœud de l'arbre pour chaque point d'où un espace $\Theta(n)$.

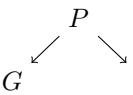
■

Proposition 27 La recherche (exacte) d'un point peut s'implémenter en pire cas en $\Theta(\log n)$.

3.2.3 Calcul du plus proche voisin

L'objectif est de cherche le point le plus proche de Q dans un *kd*-arbre. Pour cela, on réalise un *parcours en profondeur* du *kd*-arbre. Toutefois, on maintient dans une variable *bestSoFar*. le point le plus proche trouvé jusque là. S'il y a aucun chance de trouver un point proche que *bestSoFar* dans le sous-arbre couvrant, on arrête l'exploration. Ensuite, on explore en premier le sous-arbre dont la zone contient Q.

entrée : un *kd*-arbre T et un point $Q \in \mathbb{R}^d$, le meilleur point *bestSoFar* trouvé jusque là
sortie : le point dans T qui est le plus proche de Q
fonction voisinPlusProche(T, Q, *bestSoFar* := racine de T)
  si T est l'arbre vide **alors**
    renvoyer *bestSoFar*
  sinon
    T est donc de la forme



    si $d(Q, P) < d(Q, bestSoFar)$ **alors** $bestSoFar := P$
    si $Q[i] < P[i]$ **alors** $(T_1, T_2) := (G, D)$ **sinon** $(T_1, T_2) := (D, G)$
    $bestSoFar :=$ voisinPlusProche($T_1, Q, bestSoFar$)
    si $d(Q,$ hyperplan de $T) < d(Q, bestSoFar)$ **alors**
      $bestSoFar :=$ voisinPlusProche($T_2, Q, bestSoFar$)
    renvoyer *bestSoFar*

Remarque : $d(Q,$ hyperplan de $T)$ est $|Q[i] - P[i]|$.

TODO: mettre une ref ici

Proposition 28 La recherche du plus proche voisin est en $O(nd)$ dans le pire cas.

Démonstration. Au pire, on parcourt tout l'arbre qui contient n nœuds. En chaque nœud, on calcule une distance en $O(d)$. D'où le $O(nd)$. ■

Théorème 29 (admis, [FBF77], [Skr19]) La recherche du plus proche voisin est en $O(d \log n)$ en moyenne. TODO: je ne suis pas sûr à 100% car la source omet le d

DÉMONSTRATION. C'est un truc avec la distribution bêta, on verra peut-être ça en cours de proba/statistiques. Le document [Skr19] est un bon point de départ je pense. ■

En fait, en pratique c'est souvent du $O(2^d + \log n)$ TODO: cf. notes de cours de 420lects.pdf : du $\log n$ pour trouver où le plus proche est est dans l'arbre, et du 2^d pour regarder autour du point. C'est avantageux d'utiliser les kd-trees que quand il y a au moins 2^d exemples. Sinon autant utiliser la méthode naïve. Par exemple :

dimension 10	au moins 1000 exemples
dimension 20	au moins 1000000 exemples

3.3 Locality-sensitive hashing

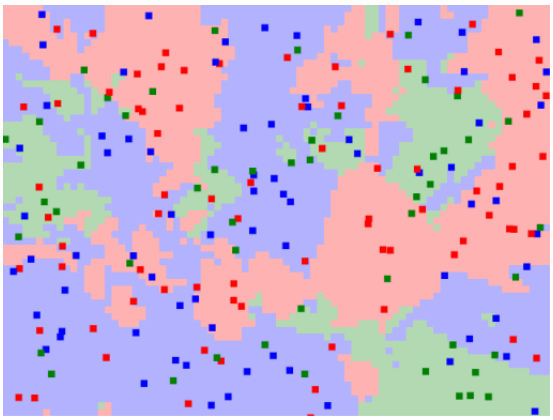
Les tables de hachage classique ne peuvent pas être utilisées car elles ne tiennent pas du tout compte de la localité. Dans une alvéole, c'est le bazar. L'idée est d'utiliser une projection sur une seule dimension, puis de discrétiser la ligne 1D en y plaçant des alvéoles. Deux points proches vont donc être mis dans la même alévole ou des alvéoles proches. Le problème est que des points éloignés peuvent aussi être placé dans la même alvéole. Du coup, la solution est de considérer *plusieurs* projections aléatoires et de les combiner. L'idée de l'algorithme pour trouver le plus proche voisin est le suivant [RN20] :

entrée : une collection de table de hachage $T_1, \dots, T_\ell$, un point $Q \in \mathbb{R}^d$
sortie : le point de $T_1 \cup T_\ell$ qui est le plus proche de Q
fonction voisinPlusProche($T_1, \dots, T_\ell$, Q)
  calculer $U := \bigcup_i$ alvéole correspondante à Q dans T_i
  calculer le plus proche voisin dans U

TODO: pourquoi l'union n'est pas trop grosse ?

4 Méthodes des plus proches voisins

☺ moins sensible au bruit
☹ besoin de choisir k (parfois on prend $k = \sqrt{n}$), c'est un hyper-paramètre



4.1 Définition

Définition 30 (apprentissage supervisé par méthode des k plus proches voisins, kNN)

- entrée : un jeu d'apprentissage $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$;
- sortie : la fonction $h : \mathcal{X} \to \mathcal{Y}$ définie par l'algorithme suivant :

fonction $h(x)$
  calculer les indices $i_1, \dots, i_k$ des k plus proches voisins x_i de x
  renvoyer la combinaison des opinions de $y_{i_1}, \dots, y_{i_k}$

Pour combiner les opinions :

- si les étiquettes sont discrètes (classification), on peut prendre l'étiquette majoritaire du multi-ensemble $\{\{y_{i_1}, \dots, y_{i_k}\}\}$. En cas de conflit, on prend au hasard une étiquette les plus présentes.
- si les étiquettes sont des réelles (régression), on peut faire la moyenne : $\frac{1}{k}\sum_{j=1}^{k} y_{i_j}$.

Remarque 31 Il y a aussi des kNN pondérés où on prend en compte les distances. On fait un vote majoritaire pondéré, ou une moyenne pondérée.

4.2 Implémentation naïve avec file de priorité

L'idée est de parcourir $x_1, \dots, x_n$. On maintient une file de priorité avec les k plus proches voisins de $x_1, \dots, x_i$. L'élément le plus prioritaire est le plus loin de Q afin de pouvoir l'éjecter au plus vite si on tombe sur x_i plus proche de Q.

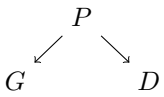
entrée : des points $x_1, \dots, x_n$ et un point $Q \in \mathbb{R}^d$
sortie : la liste des k points proches de Q parmi $x_1, \dots, x_n$
fonction voisinsPlusProches($x_1, \dots, x_n$, Q)
    $bestSoFar$:= file de priorité dans laquelle on insère $x_1, \dots, x_k$ où la priorité d'autant plus grande que la distance de x_i à Q l'est
    pour $i = k+1..n$
        si $\delta(x_{i+1}, Q) < \delta$(point le plus prioritaire de $bestSoFar, Q$)
            $bestSoFar$.défiler le point le plus prioritaire
            $bestSoFar$.insérer(x_{i+1})
    renvoyer $bestSoFar$

Proposition 32 La complexité est en $O(nd + n\log k)$.

Démonstration. On suppose que le calcul d'une distance est en $O(d)$. On peut précalculer les distances une fois pour toutes. D'où $O(dn)$. Les manipulations d'insertion et de défilement de la file sont en $O(\log k)$. D'où $O(n \log k)$ pour les opérations de la file. ■

4.3 Implémentation avec file de priorité et kd-arbre

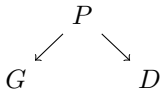
On combine l'idée d'utiliser le parcours en profondeur intelligent + une file de priorité.

entrée : un kd-arbre T et un point $Q \in \mathbb{R}^d$, une file de priorité avec les meilleurs points $bestSoFar$ trouvés jusque là
(x_i est d'autant plus prioritaire que $\delta(x_i, Q)$ est grande)
sortie : les k points dans T qui les plus proches dans Q
fonction voisinPlusProche(T, Q, $bestSoFar$:= racine de T, $bestSoFar := \emptyset$)
    si T est l'arbre vide **alors**
        renvoyer $bestSoFar$
    sinon
        T est donc de la forme 

        insérer P dans $bestSoFar$
        si $|bestSoFar| > k$ défiler de $bestSoFar$ le point le plus éloigné
        si $d(Q,P) < d(Q, bestSoFar)$ **alors** $bestSoFar := P$
        si $Q[i] < P[i]$ **alors** $(T_1, T_2) := (G, D)$ **sinon** $(T_1, T_2) := (D, G)$
        $bestSoFar$:= voisinPlusProche($T_1, Q, bestSoFar$)
        si $|bestSoFar| < k$ ou $d(Q$, hyperplan de $T) < d(Q, bestSoFar$.plus_loin) **alors**
            $bestSoFar$:= voisinPlusProche($T_2, Q, bestSoFar$)
        renvoyer $bestSoFar$

entrée : un kd-arbre T_0 et un point $Q \in \mathbb{R}^2$
sortie : les k points dans T_0 qui sont les plus proches de Q
fonction voisinPlusProche(T_0, Q)
    $bestSoFar$:= file de priorité dans laquelle on insère k points de T_0 où la priorité est d'autant plus grande que la distance de x_i à Q l'est

    fonction rec(T)

        si T n'est pas vide et est donc de la forme  **alors**

            si $d(Q$, zone correspondante à $T) \geq d(Q, bestSoFar$.élément prioritaire) **alors**
                renvoyer
            si $d(Q, P) < d(Q, bestSoFar$.élément prioritaire)
                $bestSoFar$.défiler le point le plus prioritaire
                $bestSoFar$.insérer(P)
            si Q avant P **alors**
                rec(G); rec(D)
            sinon
                rec(D); rec(G)

    rec(T_0)
    renvoyer $bestSoFar$

TODO: est-ce que la complexité est en $\log n \log k$ en moyenne ?

TODO: détailler un exemple, comme filtrage colalboratif de Azencott p. 127

5 Apprentissage par arbres de décision

TODO: reprendre avec le gros livre "Apprentissage artificiel", p. 442
☺ pas besoin de distance
☺ ils sont aussi non paramétrique, très expressif
☺ ils sont interprétables
☹ apprennent mal

5.1 Arbres de décision

Dans un premier temps, on suppose que les attributs sont tous à domaine binaire (comme fait dans [CMB18]). On discutera plus tard de comment faire si ce n'est pas le cas.

Définition 33 (arbre de décision) Un arbre de décision est un programme généré par la grammaire suivante :

$$\varphi \quad ::= \quad \textbf{renvoyer } y \quad | \quad \textbf{si } a? \textbf{ alors } \varphi \textbf{ sinon } \varphi$$

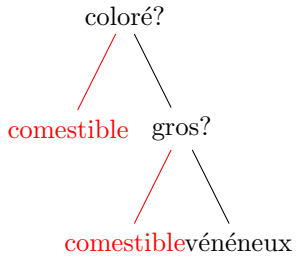
où $y \in \mathcal{Y}$ et a est un attribut.

On peut dessiner un arbre de décision par un arbre binaire fini où :

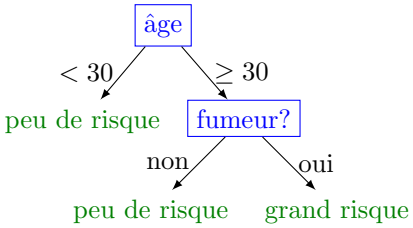
- chaque feuille est étiquetée par une classe $c \in \mathcal{Y}$
- chaque nœud interne est étiqueté par un attribut.

Exemple 34

si coloré? **alors**
    renvoyer comestible
sinon
    si gros? **alors**
        renvoyer comestible
    sinon
        renvoyer vénéneux



Exemple 35



Azencott [Aze22] décide de présenter plutôt le cas où les attributs sont quelconques. Cela complique un peu les définitions car les nœuds internes ne sont plus étiquetés par des attributs mais par : e question du type :

- $x_a = v$ (cas d'un attribut a discret)
- $x_a \geq v$ (cas d'un attribut a continu). Les valeurs v considérées sont celles qui sont entre deux valeurs consécutives $x_{i,a}$ et $x_{j,a}$ dans le dataset.

Pour des attributs à valeurs dans un ensemble fini, on pourrait brancher sur la valeur de l'attribut en faisant plusieurs niveaux. Le livre [SB14], lui fait le choix d'avoir des attributs binaires mais aussi une classification binaire. C'est compliqué à lire car il y a des 0 et 1 partout ! Dans [RN10], les attributs ont finis mais pas forcément binaires mais la décision est binaire.

🎓Construis toi-même un exemple d'un petit arbre de décision sur un sujet qui t'intéresse.

5.2 Problème algorithmique

On s'intéresse à trouver un arbre de décision qui colle aux données et qui soit petit. Par exemple, on peut demander à minimiser le nombre moyen de tests, i.e. la somme des longueurs des branches.

Définition 36 **Meilleur arbre de décision**

entrée : un dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$
sortie : un arbre de décision h consistant avec $\mathcal{D}$ et $\sum_{i=1}^{n} |branche_\tau(x_i)|$ minimal

où $|branche_\tau(x_i)|$ est le nombre de questions dans la branche que l'on suit dans τ quand on répond aux questions pour l'observation x_i.

Des fois, $\mathcal{D}$ n'est pas consistant et on veut alors minimiser $\frac{|\{i|h(x_i)\neq y_i\}|}{n}$.

Théorème 37 ([HR76]) Le (problème de décision associé) au problème de trouver le meilleur arbre de décision est NP-complet.

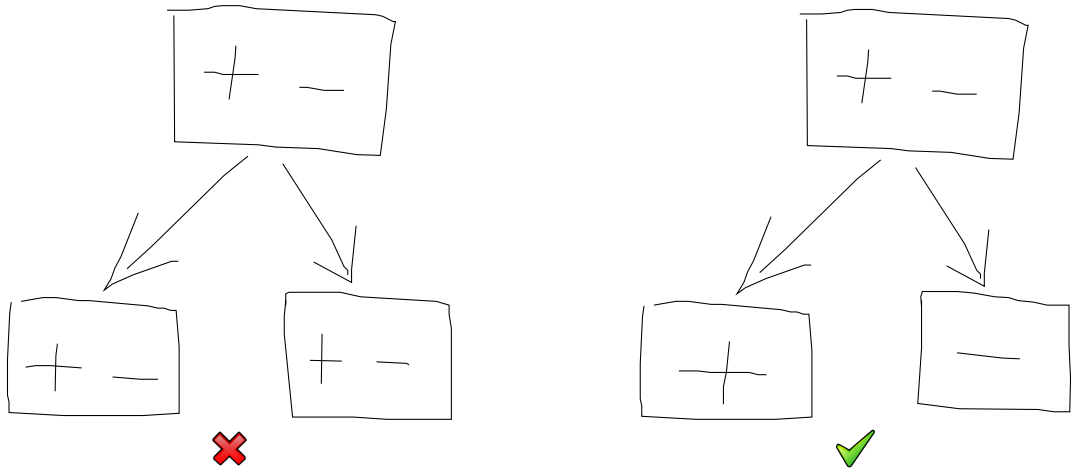
5.3 Algorithme d'apprentissage

Il est impossible de parcourir naïvement 2^{2^n} arbres qui représentent chacun une fonction booléenne. On donne donc ici un algorithme basé sur une heuristique. L'algorithme repose sur le principe de *diviser pour régner*. On cherche à trouver la meilleure question à poser, puis on divise le dataset d'apprentissage en deux, et on continue récursivement. On donne ici l'algorithme donné dans [CMB18], p. 445.

entrée : un dataset $\mathcal{D}$, une ensemble d'attributs A
sortie : un arbre de décision sur les attributs A dans pour classifier $\mathcal{D}$

fonction constructionArbre($\mathcal{D}, A$)
  si tous les exemples de $\mathcal{D}$ appartiennent à la même classe **alors**
    renvoyer feuille avec cette classe
  a := choisir le meilleur attribut par rapport à $\mathcal{D}$
  nœud := créer un nœud étiqueté par l'attribut a
  nœud.ajouterFils(constructionArbre($\mathcal{D}_{|q}$, $A \setminus \{a\}$))
  nœud.ajouterFils(constructionArbre($\mathcal{D}_{|\text{non } q}$, $A \setminus \{a\}$))
  renvoyer nœud

Les critères pour déterminer le meilleur attribut reposent sur la façon dont on sépare $\mathcal{D}$.



TODO: ajouter plein d'autres dessins avec différentes tailles
Pour cela, on définit l'espace probabilisé par rapport à $\mathcal{D}$.

5.4 Espace probabilisé

Modélisons le dataset $\mathcal{D}$ comme un espace probabilisé. On introduit deux variables aléatoires : X est la donnée et Y la classe à laquelle appartient la donnée. Ainsi par exemple :

$$\mathbb{P}_{\mathcal{D}}(Y = c) = \frac{|\{i = 1..n \mid y_i = c\}|}{n}$$

Il y a deux avantages à utiliser la théorie des probabilités. Premièrement les notations sont moins laborieuses. Désolé, mais $\frac{|\{i=1..n|y_i=c\}|}{n}$ c'est illisible en comparaison à $\mathbb{P}_{\mathcal{D}}(Y = c)$, non ? Deuxièmement, ça permet de facilement rapatrier des notions de probabilité.

5.5 Impureté de Gini

On donne maintenant une autre heuristique qui s'appelle l'impureté de Gini. ⚠ Attention, il ne faut pas confondre avec l'indice de Gini (ou coefficient de Gini) qui est un indicateur du niveau d'inégalité dans un pays etc.

L'impureté de Gini mesure de combien un élément au hasard est mal étiqueté. Le choix de l'élément est fait en tirant de manière uniforme i entre 1 et n ; puis on a notre x_i. L'étiquettage de x_i se fait en fonction de la proportion d'élements qui ne sont pas du type y_i.

Définition 38 L'impureté de Gini est :

$$Gini(\mathcal{D}) := \mathbb{P}_{\mathcal{D}}(Y \neq Y')$$

où la probabilité $\mathbb{P}_{\mathcal{D}}()$ est donné par $\mathcal{D}$, Y et Y' sont deux variables aléatoires indépendantes qui choisissent un élément au hasard de manière uniforme et renvoie la classe de cet élément.

Proposition 39

$$Gini(\mathcal{D}) := \sum_{y \in \mathcal{Y}} \mathbb{P}_{\mathcal{D}}(Y = y)(1 - \mathbb{P}_{\mathcal{D}}(Y = y))$$

Dans la définition précédente, avec une probabilité $\mathbb{P}(Y = y)$ on tire un élément de classe y. Puis, la proportion de ne pas être un élément de classe y est $1 - \mathbb{P}(Y = y)$.

Maintenant, on va regarder ce que ça donne quand on branche sur l'attribut a. On prend alors la moyenne pondérée sur les valeurs v de l'attribut a :

Définition 40

$$Gini(\mathcal{D}, a) = \sum_{v \in Dom_{X_a}} \mathbb{P}_{\mathcal{D}}(X_a = v) Gini(\mathcal{D}_{X_a = v}).$$

choisir a qui minimise $Gini(\mathcal{D}, a)$

5.6 Entropie comme critère

On veut que quand on splitte par rapport à un attribut, ça soit de plus en plus ordonné.

J'ai une surprise pour vous

L'entropie mesure la quantité de désordre.
Par exemple, si j'ai un dataset de champignon et qu'ils sont tous toxiques, je n'ai aucune surprise à en trouver un toxique. C'est juste la normalité des choses. Par contre, si j'ai un grand désordre ; découvrir la classe d'un champignon est toujours une surprise pour moi.
Ainsi, on définit la notion de surprise. Si E est un événement aléatoire, alors on définit la surprise d'avoir E comme

$$surprise(E) := \log_2 \frac{1}{\mathbb{P}(E)} = -\log_2 \mathbb{P}(E).$$

Plus E est probable, moins il est surprenant. Pourquoi $\log_2$? Parce que le logarithme transforme les multiplications en additions. Et donc si E et F sont indépendants, alors $surprise(E \cap F) = surprise(E) + surprise(F)$.

Définition 41

$$\begin{aligned} \mathbb{H}(\mathcal{D}) &:= \text{moyenne des surprises de } Y = y \\ &= -\sum_{y \in Dom_Y} \mathbb{P}(Y = y) \times \log_2(\mathbb{P}(Y = y)) \mathbb{H}(Y) \end{aligned}$$

où Y est une variable aléatoire qui choisit aléatoirement uniformément un élément dans $\mathcal{D}$ et donne sa classe.

TODO: et si c'est de la régression

Définition 42 (entropie conditionnelle)

$$\mathbb{H}(\mathcal{D} \mid a) = \sum_{v \in Dom_{X_a}} \mathbb{P}_{\mathcal{D}}(X_a = v) \mathbb{H}(\mathcal{D}_{X_a = v}).$$

choisir a qui minimise $\mathbb{H}(\mathcal{D} \mid a)$

choisir a qui minimise $\mathbb{H}(Y|X_a)$ où $\mathbb{P}()$ est $\mathbb{P}_{\mathcal{D}}()$

Dans la littérature, on trouve la notion de *information gain* qui est $IG(\mathcal{D}, a) := \mathbb{H}(Y) - \mathbb{H}(Y|X_a)$ où les entropies sont calculées par rapport à l'espace probabilisé de $\mathcal{D}$.

TODO: la suite c'est la version dans understanding machine learning

TODO: pourquoi les gens définissent le gain alorsque manipuler directement $Gini(\mathcal{D}, a)$ c'est plus simple

5.7 Se repérer dans la littérature

Voici un algorithme générique, basé sur l'algorithme CART (Classification And Regression Tree) de Breiman et al. Pour l'algorithme CART, on utilise l'indice de Gini, autrement dit on prend : $C(\alpha) = 2\alpha(1-\alpha)$. Pour ID3 et C4.5, on utilise les même définitions mais avec l'entropie. On prend le gain d'information : $C(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$.

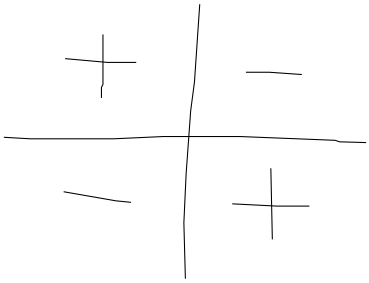
Contrairement à l'erreur, le gain d'information et l'indice de Gini sont concaves et sont des bornes supérieures de l'erreur. C'est cool ! TODO: étudier pourquoi c'est cool

5.8 Arbres trop gros !

Les arbres appris sont trop gros ! Il y a de l'overfitting. Pour combattre l'overfitting, on fait de l'élaguage de l'arbre appris.

5.8.1 Solution naïve : early-stopping

Une solution naïve pourrait être d'arrêter la construction de l'arbre si notre critère (entropie ou impureté de Gini) est trop peu informatif. Mais le problème que ça arrive de pas avoir de bon attribut à un moment donné. Par exemple si on veut apprendre $p \oplus q$ où $\oplus$ est le XOR, on est obligé de brancher sur l'un des attributs, disons p, qui n'est pas informatif. Puis un second branchement sur q permet de trancher.



5.8.2 Pruning

TODO: section 18.2.2 TODO: regarder le bouquin de prépa

Le but est d'éliminer les nœuds qui ne sont pas pertinents. Mais nous le faisons du bas vers le haut. Ainsi, ça va bien marcher sur l'exemple du XOR.

6 Random forest

Malheureusement, les arbres de décision sont des apprenants faibles : en pratique, les résultats sont à peine meilleur que du random. Une solution est alors de faire de l'ensemble learning.

Random forest c'est apprendre B arbres de décision :

1. $\mathcal{D}' :=$ Tirer n observations (avec remise) de $\mathcal{D}$ (Tree bagging)
2. Choisir un sous-ensemble A' d'attributs au hasard (typiquement $\sim \sqrt{d}$) (feature sampling)
3. Lancer l'algorithme d'apprentissage d'un arbre de décision sur $\mathcal{D}'$ en se restreignant aux attributs dans A'

Prédiction avec random forest :

1. calculer les prédictions des B arbres sur x
2. faire vote majoritaire

7 Réseaux de neurones

☺ modèle non-linéaire, expressif
☹ ça converge souvent vers un minimum local
☹ problème de disparition et d'explosion du gradient
☹ problème de saturation du réseau

7.1 Définitions

Définition 43 Un circuit est graphe $G = (V, E)$ orienté acyclique où :

- les sommets dans V sont appelés des portes/neurones ;
- les sommets sans prédécesseur sont appelé des entrées. On les notes $x_1, \dots, x_n$.
- les sommets sans successeur s'appellent des sorties.
- chaque sommet j qui n'est pas une entrée est étiqueté par une fonction $\mathbb{R}^{pred(j)} \to \mathbb{R}$.

Exemple 44 (circuit arithmétique) TODO: faire un dessin

Parfois, un circuit est organisé par couches, c'est-à-dire $V = \bigsqcup_{t=0}^{T} V_t$ et :

- V_0 = ensemble des neurones d'entrées
- V_T = ensemble des neurones de sortie
- Chaque arc dans E connecte un sommet d'un certain V_t à V_{t+1}.

Les couches $V_1, \dots, V_{T-1}$ s'appelle souvent les couches cachées. Mais en fait, on s'en fiche ici.

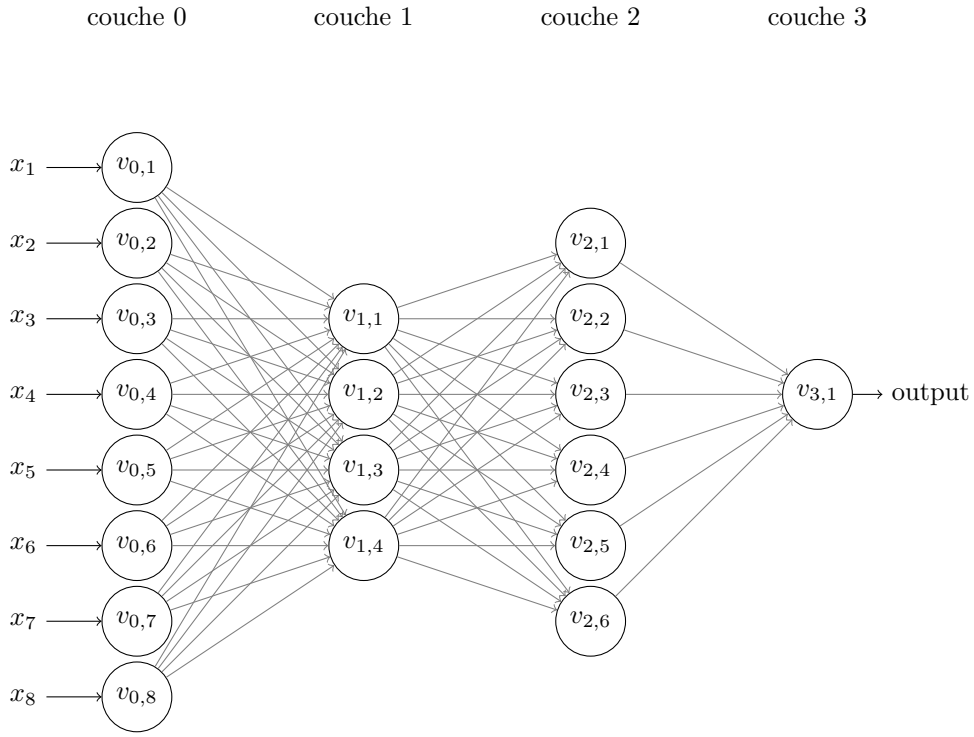
Définition 45 Un *réseau de neurones* est un circuit où chaque neurone j qui n'est pas une entrée est équipé d'une fonction :

$$(out_i)_{i \in pred(j)} \mapsto \sigma \left(\sum_{i \in pred(j)} w_{ij} out_i \right)$$

où σ est une fonction d'activation et les w_{ij} sont des poids dans $\mathbb{R}$.

Un réseau de neurones organisé par couches et où il y a tous les arcs d'une couche à une autre, s'appelle un multi-layer perceptron (MLP).

Exemple 46 Voici une représentation graphique d'un réseau de neurones. Il s'agit ici d'un MLP :



Dans la suite, pour simplifier le discours, on suppose que les réseaux de neurones n'ont qu'une seule sortie. On note w la collection de tous les paramètres. Un réseau de neurones représente une fonction $f_w : \mathbb{R}^n \to \mathbb{R}$.

On définit les entrées/sorties des neurones de la façon suivante :

- $out_j(x) = x[j]$ si j est un neurone d'entrée, on vient lire l'entrée.
- pour tout $j \geq 1$,
  - $out_j(x) = \sigma\left(\sum_{i \in pred(j)} w_{ij} out_i\right)$.

  Si on note $final$ le neurone final, alors la valeur $f_w(x)$ est la valeur de out_{final} quand les entrées sont données $x = (x[1], \ldots, x[d])$.

Exemple 47 Le réseau de neurones ci-dessus représente la fonction $f_w(x_1, x_2, \ldots, x_8) = \sigma\left(\sum_{k=1}^{6} w_{2,k,3,1} out_{2,k}\right)$ où :

- $out_{2,k} = \sigma\left(\sum_{j=1}^{4} w_{1,j,2,k} out_{1,j}\right)$
- $out_{1,j} = \sigma\left(\sum_{i=1}^{8} w_{0,i,1,j} out_{0,i}\right)$
- $z_{0,j} = x[j]$.

7.2 Problèmes d'entraînement

Voici l'erreur qui mesure de combien le réseau de neurone se trompe sur une observation x de classe y. C'est une erreur quadratique : c'est la différence entre le vrai label y et le résultat du réseau $f_w(x)$. Le facteur $1/2$ c'est juste pour faire joli, quand on va dériver (mais vous avez l'habitude avec l'énergie cinétique, ou d'autres expressions qui ont un $1/2$ comme ça).

Définition 48 (loss/cost/error function) $E(x, y, w) = \frac{1}{2}(y - f_w(x))^2$.

On se fixe une architecture de réseau ; i.e. le graphe orienté acyclique est fixé. Mais il s'agit de trouver les poids tel que f_w soit un bon modèle.

Définition 49 (problème d'apprentissage supervisé)

- entrée : un jeu d'apprentissage $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$;
- sortie : des poids w (i.e. une fonction f_w) tels que $\sum_{i=1}^{n} E(x_i, y_i, w)$ minimal.

Ce problème d'optimisation est très difficile à résoudre. TODO: donner sa complexité ? Donc en fait on fait une descente de gradient plutôt. On fait même une descente de gradient pour chaque donnée. Et tant pis si c'est un minimum local.

L'idée de notre descente de gradient est quand on reçoit (x, t) on ajuste les poids w du réseau en faisant :

$$w := w - \alpha \nabla_w E(x, y, w)$$

où α est le pas d'apprentissage (*learning rate*) et $\nabla_w E(x, y, w)$ est le vecteur gradient en (x, y, w) mais dérivant juste par rapport aux poids w. Le vecteur gradient est formée des dérivées partielles $\frac{\partial E}{\partial w_{ij}}(x, y, w)$.

Définition 50 (vecteur gradient)

$$\nabla_w E(x, y, w) := \left(\frac{\partial E}{\partial w_{ij}}(x, y, w)\right)_{i,j \text{neurones avec } i \to j}.$$

Définition 51 (calcul du gradient) Le problème du calcul du gradient est :

- entrée : x l'observation, y le vrai label, w les poids actuel
- sortie : le vecteur gradient $\nabla_w E(x, y, w)$

7.3 Notations

On note i, j, k des neurones. Par convention, on suppose que quand l'ordre alphabétique est croissant, alors on va vers la droite dans le réseau.

On note :

$final$	neurone final (où on a la sortie du réseau)
out_i	la sortie du neurone i
out_{final}	la sortie du réseau = sortie du neurone final
$comb_i$	la combinaison linéaire au neurone i
σ	la fonction d'activation (la même en tous les neurones)
E	la fonction d'erreur (on écrit E au lieu de $E(x, y, w)$)
w_{ij}	poids du neurone i au neurone j

7.4 Exprimer le vecteur gradient avec des δ_j

On introduit une notation spéciale δ_j pour la dérivée partielle de E par rapport à $comb_j$. On le fait car on calculera plus tard les δ_j par programmation dynamique (backpropagation).

Définition 52 Pour tout neurone j,

$$\delta_j = \frac{\partial E}{\partial comb_j}.$$

On rappelle le **théorème de dérivation des fonctions composées**, i.e.

$$\frac{d}{dx} f(g_1(x), \dots, g_k(x)) = \sum_{i=1}^{k} \left(\frac{d}{dx} g_i(x) \right) D_i f(g_1(x), \dots, g_k(x))$$

où $D_i f$ est la dérivée partielle de f par rapport à la i-ième coordonnée. On met un (*) quand on l'utilise. Je vous invite comme exercice (lourdingue !) à bien détailler les fonctions f et $g_1, \dots, g_k$ pour chacune des applications du théorème.

La proposition suivante explique que le calcul du vecteur gradient s'exprime avec les δ_j.

Proposition 53

$$\frac{\partial E}{\partial w_{ij}} = \delta_j out_i$$

Démonstration. On utilise (*) :

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial comb_j} \frac{\partial comb_j}{\partial w_{ij}}$$

Pour (*), $comb_j$ est la seule combinaison linéaire où w_{ij} intervient donc les autres dérivées partielles sont nulles.

Lemme 54

$$\frac{\partial comb_j}{\partial w_{ij}} = out_i$$

Démonstration. Car $comb_j = \sum_i w_{ij} out_i$. Aux yeux de la variable w_{ij} tout est constant sauf $w_{ij} out_i$ qui ressemble à $w_{ij} \times cte$. La constante cte est donc la dérivée cherchée. ■

■

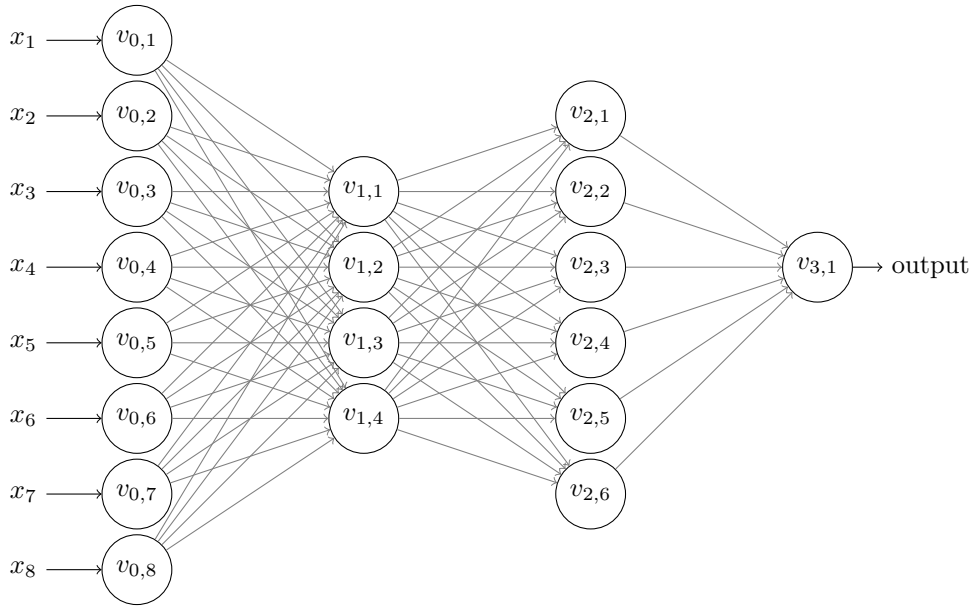
entrée : une observation x, son vrai label y, les poids actuel w
sortie : $\nabla_w E(x, y, w)$ le vecteur gradient de la fonction erreur par rapport aux poids
fonction calculerVecteurGradientErreur(x, y, w)
  calculer $comb_j$, out_j pour tout neurone j à partir de l'entrée x et les poids w (propagation avant)
  calcul des δ_j (c'est ce que l'on appelle propagation arrière, on verra pourquoi dans la section suivante)
  pour tout neurone i, j avec $i \to j$
    calculer $\frac{\partial E}{\partial w_{ij}}$ qui est $\delta_j out_i$
  renvoyer $\left(\frac{\partial E}{\partial w_{ij}}\right)_{i,j \text{neurones avec } i \to j}$

7.5 Algorithme de backpropagation : programmation dynamique pour calculer les δ_j

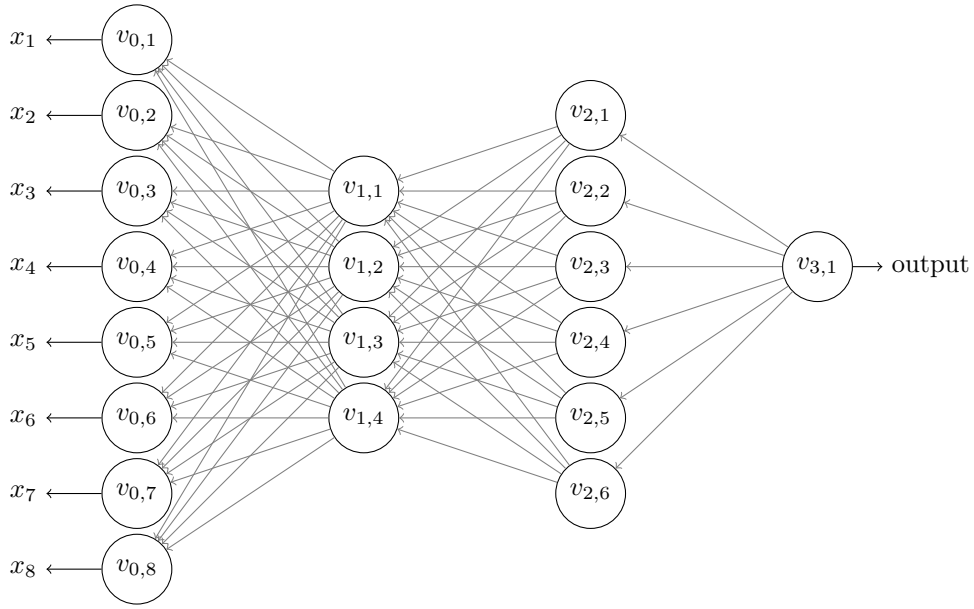
De manière étonnante, le calcul des δ_j se fait par *programmation dynamique* de droite à gauche dans le réseau. Qui dit programmation dynamique, dit relation de récurrence. On traite d'abord le cas de base puis la relation de récurrence proprement dite. TODO: Bishop dit $\delta_{final} = f_w(x) - y$, c'est nimporte quoi p. 243....

Le graphe des sous-problèmes est le réseau de neurones mais où les arcs vont dans l'autre sens.

Exemple 55 Si le réseau est :



alors le graphe des sous-problèmes est :



où en chaque neurone, le sous-problème est de calculer δ_j.

TODO: regarder Cornuéjols p. 286

Le cas de base a lieu pour le neurone final. C'est là qu'il y a "le sous-problème" le plus petit. La valeur δ_{final} pour le neurone final $final$ est donné par la proposition suivante.

Proposition 56 (cas de base des δ)

$$\delta_{final} = (f_w(x) - y) \times \sigma'(comb_{final})$$

Démonstration.

$$\begin{aligned}\delta_{final} &= \frac{\partial E}{\partial comb_{final}} \\ &= \frac{\partial \frac{1}{2}(\sigma(comb_{final}) - y)^2}{\partial comb_{final}} \\ &= (\sigma(comb_{final}) - y) \times \sigma'(comb_{final})\end{aligned}$$

■

Voici la relation de récurrence qui exprime δ_j en fonction de valeurs δ_k où k sont des neurones dans la couche suivante. Les δ_k correspondent donc à des problèmes "plus petits".

Proposition 57 (relation de récurrence des δ)

$$\delta_j = \sigma'(comb_j) \sum_{k|j\to k} w_{jk}\delta_k$$

Démonstration.

$$\begin{aligned}\delta_j &= \frac{\partial E}{\partial comb_j} \\ &= \sum_{k|j\to k} \frac{\partial E}{\partial comb_k}\frac{\partial comb_k}{\partial comb_j}(*) \\ &= \sum_{k|j\to k} \delta_k \frac{\partial comb_k}{\partial comb_j}\end{aligned}$$

Pour calculer $\frac{\partial comb_k}{\partial comb_j}$, écrivons l'expression de $comb_k$ pour y voir un peu plus clair :

$$comb_k = \sum_{j'|j'\to k} w_{j'k} out_{j'} = \sum_{j'|j'\to k} w_{j'k}\sigma(comb_{j'}).$$

Ainsi,

$$\frac{\partial comb_k}{\partial comb_j} = w_{jk}\sigma'(comb_j).$$

■

7.6 Wrap-up : pseudo-code de l'algorithme de backpropagation

entrée : une observation x, son vrai label y, les poids actuel w
sortie : $\nabla_w E(x, y, w)$ le vecteur gradient de la fonction erreur par rapport aux poids
fonction calculerVecteurGradientErreur(x, y, w)
  calculer $comb_j$, out_j pour tout neurone j à partir de l'entrée x et les poids w (propagation avant)
  $\delta_{final} := (f_w(x) - y) \times \sigma'(comb_{final})$
  pour tout neurone j des derniers vers les premiers, sauf le final
    $\delta_j := \sigma'(comb_j) \sum_{k|j\to k} w_{jk}\delta_k$
  pour tout neurone i, j avec $i \to j$
    calculer $\frac{\partial E}{\partial w_{ij}}$ qui est $\delta_j out_i$
  renvoyer $\left(\frac{\partial E}{\partial w_{ij}}\right)_{i,j\text{neurones avec } i\to j}$

References

[Aze22] Chloé-Agathe Azencott. *Introduction au Machine Learning-2e éd.* Dunod, 2022.

[CMB18] Antoine Cornuéjols, Laurent Miclet, and Vincent Barra. *Apprentissage artificiel: Deep learning, concepts et algorithmes*. Eyrolles, 2018.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[HR76] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Inf. Process. Lett.*, 5(1):15–17, 1976.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[SB14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.

[Skr19] Martin Skrodzki. The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time. *CoRR*, abs/1903.04936, 2019.

Classes P et NP

François Schwarzentruber

1 Motivation

Prenons l'exemple du problème de coloration d'un graphe non orienté, disons avec 3 couleurs.

3-COLORATION (problème de recherche)

entrée : Un graphe non orienté $G = (S, A)$;
sortie : une coloration $c : S \to \{1, 2, 3\}$ avec pour tout $(s, t) \in A$, $c(s) \neq c(t)$ s'il en existe une ; IMPOSSIBLE sinon.

Ce problème intervient dans plusieurs applications. Vous avez des tâches à faire. Les sommets sont les tâches. On met une arête quand deux tâches ne sont pas faites en même temps. Il faut trouver un emploi du temps. Les couleurs sont alors les créneaux horaires.

De la même façon, il faut stocker les valeurs d'un programme dans des registres. Mais lesquels ? Les sommets sont les valeurs à stocker. On met une arête quand on a besoin des deux valeurs en même temps (elles doivent donc être dans des registres différents). Les couleurs sont les registres.

État des connaissances de l'humanité en 2023.

- Pas d'algorithme très efficace connu (i.e. rapide dans tous les cas, y compris les pires)
- Des algorithmes de type backtracking (essai - erreur) en temps exponentielle en $|G|$
- des algorithmes avec utilisation fine de la programmation dynamique. Le meilleur en $O(1.7272^{\#\text{sommets}})$ de 2007.
- Si le nombre de couleurs n'est pas fixé, on a des algorithmes d'approximation avec des facteurs bizarres. Voir Wikipedia.

Bref, 3-coloration a l'air d'être un problème difficile algorithmiquement parlant. Dans ce cours, nous allons donner aborder la théorie de la complexité. Elle donne un argument *théorique* pour dire que, oui, le problème de 3-coloration est algorithmiquement et intrinsèquement difficile. Il est NP-dur (et on va voir ce que ça veut dire dans ce cours).

Imaginez vous avez trouver un algorithme en $O(1.717171^{\#sommets})$ pour 3-coloration. Vous le publiez. Vous n'avez pas lu toute la littérature scientifique. Si cela se trouve, il y a un algorithme efficace quelque part.

Alors la théorie de la complexité, vous êtes 'rassurée'. Le problème de 3-coloration que j'ai étudié est NP-dur. On a un argument solide, une publication à citer qui confirme que le problème est difficile. A priori, il n'y a pas d'algorithme efficace (on en a pas la certitude car P = NP est un problème ouvert, mais au moins on en a quelque chose à dire !). Si la théorie de la complexité n'existait pas, on Souvent dans les papiers de recherche, on étudie la complexité théorique. Ca donne de la pertinence

2 Problèmes de décision

Définition 1 (problème de décision)
Un *problème de décision* est une fonction qui à une entrée x associe oui ou non. L'entrée x s'appelle une *instance* : c'est une suite de bits. La *taille* $|x|$ d'une instance x est le nombre de bits dans x. Une instance d'un problème est *positive* si la réponse associée est oui. Elle est dite *négative* sinon.

Dans la suite, on ne considère que des problèmes de décision. En voici des exemples :

Exemple 2 **PRIMES**

entrée : un entier (en représentation binaire)
sortie : oui, si l'entier est premier ; non, sinon.

Exemple 3 **Accessibilité**

entrée : un graphe orienté, un sommet source s et un sommet destination t
sortie : oui, s'il existe un chemin de s à t dans le graphe G ; non sinon.

Exemple 4 **Arbre couvrant de poids minimum**

entrée : un graphe non orienté connexe pondéré $G = (S, A, poids)$, un seuil k
sortie : oui, s'il existe un arbre couvrant de poids $\leq k$; non sinon.

Exemple 5 **Distance d'édition**
entrée : deux mots x et y, un seuil k
sortie : oui, si la distance d'édition entre x à y est $\leq k$; non sinon.

Exemple 6 **Couplage maximum biparti**
entrée : un graphe G biparti, un seuil k
sortie : oui, s'il existe un couplage dans G de cardinal $\geq k$; non sinon.

Exemple 7 **Flot maximal**
entrée : un réseau G, un seuil k
sortie : oui, s'il existe un flot dans G de valeur $\geq k$; non sinon.

Exemple 8 **Flot maximal à valeurs entières**
entrée : un réseau G, un seuil k
sortie : oui, s'il existe un flot à valeurs entières dans G de valeur $\geq k$; non sinon.

Exemple 9 **CLIQUE**
entrée : Un graphe non orienté $G = (S, A)$, un entier k ;
sortie : oui si il y a une clique (un sous-ensemble de sommets tous reliés entre eux) de taille k dans G ; non sinon.

Exemple 10 2-**COLORATION**
entrée : Un graphe non orienté $G = (S, A)$;
sortie : oui si G est 2-coloriable, i.e. il existe une coloration $c : S \to \{1, 2\}$ avec pour tout $(s, t) \in A$, $c(s) \neq c(t)$; non sinon.

Exemple 11 3-**COLORATION**
entrée : Un graphe non orienté $G = (S, A)$;
sortie : oui si G est 3-coloriable, i.e. il existe une coloration $c : S \to \{1, 2, 3\}$ avec pour tout $(s, t) \in A$, $c(s) \neq c(t)$; non sinon.

Exemple 12 **SAT**
entrée : une formule de la logique propositionnelle φ
sortie : oui, si la formule φ est satisfiable, non sinon.

Exemple 13 **Sac à dos avec répétition (sans répétition)**
entrée : une collection d'objets $(p_i, v_i)_{i=1..n}$, un poids maximal P, un seuil k
sortie : oui, s'il existe n_i dans $\mathbb{N}$ (dans $\{0, 1\}$) avec $\sum_{i=1}^{n} n_i v_i \geq k$ avec $\sum_{i=1}^{n} n_i p_i \leq P$; non sinon.

Une instance x est une instance $\left\{ \begin{array}{l} \textit{positive} \\ \textit{négative} \end{array} \right.$ de **A** si la réponse associée est $\left\{ \begin{array}{l} \text{oui} \\ \text{non} \end{array} \right.$.

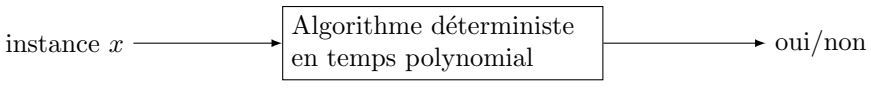
3 Classe P

Définition 14 (classe P)
P est la classe des problèmes décidés par un algorithme **déterministe** en temps polynomial.

- P est la classe des problèmes décidés par un machine de Turing **déterministe** en temps polynomial.

- Un problème **A** est dans P s'il existe un algorithme **déterministe** a tel que :
  1. pour toute instance x, x est positive pour **A** ssi $a(x)$ répond oui
  2. pour toute instance x, $a(x)$ s'exécute en temps $poly(|x|)$.

- Un problème **A** est dans P s'il existe un algorithme **déterministe** a, un polynôme p tels que :
  1. pour toute instance x, x est positive pour **A** ssi $a(x)$ répond oui
  2. pour toute instance x, le temps d'exécution de $a(x)$ majorée par $p(|x|)$



Exemple 15 **Accessibilité**, **Arbre couvrant de poids minimum**, **Distance d'édition**, **Flot maximal**, **Flot maximal à valeurs entières**, **Couplage maximum** sont dans P.

Thèse 16 (thèse d'Edmonds-Cobham) Un problème dans P est facile.

- 👍 La plupart des algorithmes en temps polynomial sont en $O(n)$, $O(n^2)$, $O(n \log n)$, donc efficaces.
- 👍 Des fois, on utilise des algorithmes en temps exponentiels mais efficaces en pratique bien que le problème soit dans P (e.g.programmation linéaire réelle)
- 👍 Stabilité par boucle for

```
pour i := 1 à n
|   algoefficace(i)
```

- 👍 Stabilité par modèle de calcul (programmes Java, machines de Turing, machines RAM, etc.)
- 👎 Un algorithme qui réalise n^{100} opérations est inefficace en pratique.
- 👎 Complexité pire cas.
- 👎 Quid des aspects quantiques, probabilistes ?

4 Classe Exptime

Plusieurs problèmes comme **SAT**, **3-COLORATION**, **Sac à dos avec répétition**, **Sac à dos sans répétition** admettent des algorithmes en temps exponentiel. On dit qu'ils appartiennent à la classe Exptime.

Définition 17 (classe Exptime) La classe Exptime est la classe des problèmes de décision qui admettent un algorithme **déterministe** en temps exponentiel en la taille de l'entrée.

▶ Exptime est la classe des problèmes décidés par un machine de Turing **déterministe** en temps exponentiel.

▶ Un problème **A** est dans Exptime s'il existe un algorithme **déterministe** a tel que :

1. pour toute instance x, x est positive pour **A** ssi $a(x)$ répond oui
2. pour toute instance x, $A(x)$ s'exécute en temps $2^{poly(|x|)}$.

▶ Un problème **A** est dans Exptime s'il existe un algorithme **déterministe** a, un polynôme p tels que :

1. pour toute instance x, x est positive pour **A** ssi $a(x)$ répond oui
2. pour toute instance x, le temps d'exécution de $a(x)$ majorée par $2^{p(|x|)}$

5 Algorithmes non-déterministes

En fait, on peut être plus fin que cela. La classe Exptime est bien trop grosse ! Elle contient des problèmes bien plus difficiles (comme par exemple, savoir si le joueur blanc a une stratégie gagnante depuis un certain plateau d'échecs généralisés). Des problèmes comme 3-coloration ont une forme spécifique. Il s'agit de trouver une solution de taille polynomiale où la vérification est en temps polynomial.

Bref, les problèmes qui nous intéressent admettent un jeu à un joueur : et en y jouant, le joueur a une stratégie gagnante ssi l'instance est positive. De plus le jeu ne dure pas longtemps : un temps polynomial en la taille de l'instance au plus.

Exemple 1

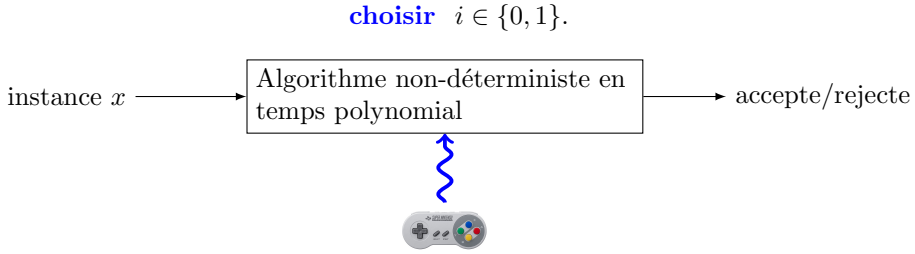
```
procédure 3-coloration(G)
|   pour s ∈ S
|   |   choisir c[s] dans {0, 1, 2} ;
|   si c est une 3-coloriation
|   |   accept (gagné)
|   sinon
|   |   reject (perdu)
```

Exemple 2

```
procédure sat(φ)
|   pour toute proposition atomique p dans φ
|   |   choisir ν[p] dans {faux, vrai} ;
|   si ν ⊨ φ
|   |   accept (gagné)
|   sinon
|   |   reject (perdu)
```

Dans le jargon de la théorie de la complexité, un tel jeu s'appelle un algorithme non-déterministe.

Définition 18 (algorithme non-déterministe) Un algorithme **non-déterministe** est similaire à un algorithme déterministe, mais peut contenir des **choix non-déterministes** d'un bit par un joueur :



Remarque 19 Un algorithme déterministe est un algorithme non-déterministe qui ne contient pas de choix pour le joueur.

Définition 20 (acceptation d'une instance) Un algorithme non-déterministe a accepte une instance x s'il existe une exécution acceptante de $a(x)$.

Définition 21 (décision d'un problème de décision) Un algorithme non-déterministe décide un problème de décision **A** si :
- toute exécution de $a(x)$ s'arrête ;
- une instance x est positive ssi $a(x)$ admet une exécution acceptante.

6 Classe NP

On est prêt maintenant à définir une classe qui est incluse dans EXPTIME et qui capture mieux les problèmes de recherche d'une solution comme **3-coloration**.

6.1 Définition historique avec les algorithmes non-déterministes

Définition 22 (NP)
NP est la classe des problèmes décidés par un algorithme **non-déterministe** en temps polynomial.

▶ NP est la classe des problèmes décidés par une machine de Turing **non-déterministe** en temps polynomial.

▶ Un problème **A** est dans NP s'il existe un algorithme **non-déterministe** a tel que
1. pour toute instance x, x est positive pour **A** ssi **il existe une exécution acceptante** de $a(x)$
2. Toutes les exécutions de $a(x)$ sont de longueur $poly(|x|)$.

▶ Un problème **A** est dans NP s'il existe un algorithme **non-déterministe** a et un polynôme p tels que
1. pour toute instance x, x est positive pour **A** ssi **il existe une exécution acceptante** de $a(x)$
2. Toutes les exécutions de $a(x)$ sont de longueur majorée par $p(|x|)$.

Proposition 23 **3-COLORATION** est dans NP.

Proposition 24 P $\subseteq$ NP.

Proposition 25 NP $\subseteq$ EXPTIME.

6.2 Définition équivalente avec les certificats

Définition 26 (NP) Un problème est dans NP s'il existe un algorithme déterministe v qui vérifie qu'une **solution** est correcte en temps polynomial.

▶ Un problème **A** est dans NP s'il existe un algorithme déterministe v tel que :
1. pour toute instance x,
   x est positive pour **A** ssi **il existe un mot** c de longueur $poly(|x|)$ tel que $v(x,c)$ répond oui
2. pour toute instance x, pour tout mot $c \in \{0,1\}^{poly(|x|)}$, $v(x,c)$ s'exécute en temps $poly(|x|)$.

▶ Un problème **A** est dans NP s'il existe un algorithme déterministe v, deux polynômes p, q tels que :

1. pour toute instance x,
   x est positive pour **A** ssi **il existe un mot** c de longueur $q(|x|)$ tel que $v(x, c)$ répond oui
2. pour toute instance x, pour tout mot $c \in \{0,1\}^{q(|x|)}$, le temps d'exécution de $v(x, c)$ est majoré par $p(|x|)$.

▶ Un problème **A** est dans NP s'il peut s'écrire sous la forme suivante :

A
entrée : une instance x
sortie : **existe-t-il un suite de bits** c de longueur $poly(|x|)$ tel qu'une propriété $\mathcal{V}(x, c)$ soit vraie ?

avec $\mathcal{V}(instance, c)$ vérifiable en temps $poly(|x|)$.

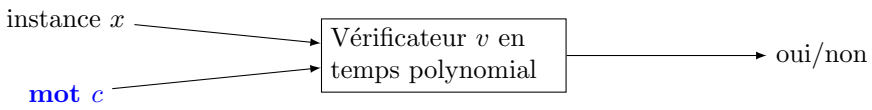
▶ Un problème **A** est dans NP s'il peut s'écrire sous la forme suivante :

A
entrée : une instance x
sortie : **existe-t-il un suite de bits** c de longueur $poly(|x|)$ tel que $v(x, c)$ renvoie vrai ?

avec $v(x, c)$ un algorithme en temps $poly(|x|)$.

Définition 27 (vérifieur ou vérificateur)
L'algorithme v s'appelle un vérifieur ou un vérificateur.

Définition 28 (certificat)
Un mot c tel que $v(x, c)$ renvoie vrai est un **certificat** pour x.

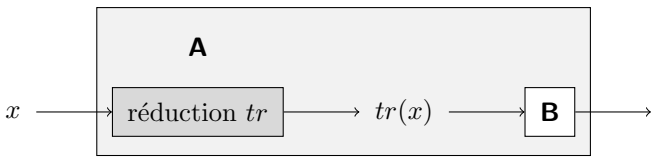


Proposition 29 La définition de NP avec algorithme non-déterministe et celle avec vérificateur sont équivalentes.

? Est-ce que P = NP ? Est-ce que P ≠ NP ?

7 Réduction en temps polynomial

On formalise ici la relation de facilité entre problèmes : **A** soit plus facile que **B**.
Le problème **A** se réduit à **B** si je peux décider **A** en transformant une **A**-instance en une **B**-instance, puis en utilisant un algorithme pour **B**. La transformation se calcule en temps polynomial.

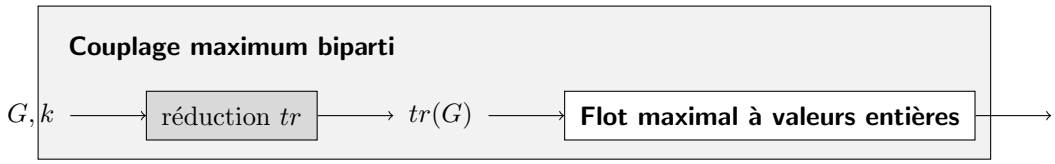


Définition 30 (Réduction en temps polynomial) Une *réduction en temps polynomial* de **A** à **B** est une fonction tr, qui, à toute instance x de **A**, associe une instance $tr(x)$ de **B**, telle que

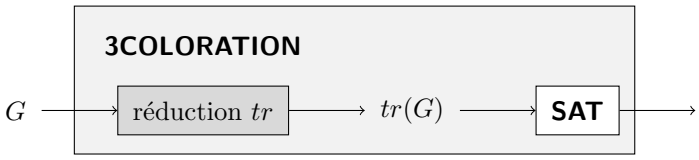
1. pour toute instance x de **A**, x est une instance positive de **A** ssi $tr(x)$ est une instance positive de **B** ;
2. $tr(x)$ calculable en temps $poly(|x|)$.

Définition 31 On dit que **A** se réduit à **B** s'il existe une réduction en temps polynomial de **A** à **B**.

Exemple 32



Exemple 33



On note **A** $\leq_p$ **B** pour dire que **A** se réduit à **B** en temps polynomial. La relation $\leq_p$ signifie intuitivement 'est plus simple que', 'se réduit en temps polynomial à'. On a donc :

— **Couplage maximum biparti** $\leq_p$ **Flot maximal à valeurs entières**
— **3COLORATION** $\leq$ **SAT**.

La relation $\leq_p$ est réflexive et transitive. C'est un *préordre*.

Proposition 34

— Si **A** se réduit à **B** et **B** dans P, alors **A** est dans P.
— Si **A** se réduit à **B** et **B** dans NP, alors **A** est dans NP.

8 NP-difficulté

Définition 35 (NP-difficile) Un problème est NP-*difficile* si tout problème de NP s'y réduit en temps polynomial.

Proposition 36 Si un problème NP-difficile est dans P alors P = NP.

Proposition 37 Si **A** se réduit polynomialement à **B** et que **A** est NP-difficile alors **B** est NP-difficile.

9 Problèmes NP-complets

Définition 38 (NP-complet) Un problème est NP-*complet* s'il est dans NP et NP-difficile.

10 Notes bibliographiques

La classe NP a été définie initialement avec des machines de Turing non-déterministes par Cook en 1971 [Coo71]. Le non-déterminisme est aussi présent dans d'autres modèles de calcul : automates, automates à pile. Karp a donné et montré que plusieurs problèmes sont NP-complets [Kar72]. On trouve une grande liste de problèmes NP-complets dans [GJ79].

Références

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

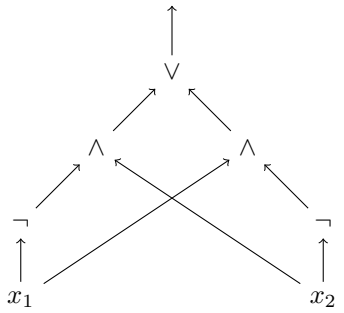
ALGO1 – NP-complétude

François Schwarzentruber

1 Théorème de Cook

On donne ici une démonstration alternative du théorème de Cook *sans* machines de Turing. Cela sera abordé en TD.

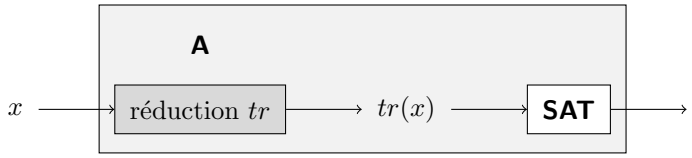
CIRCUIT-SAT
entrée : un circuit booléen C
sortie : existe-t-il des valeurs booléennes $x \in \{0,1\}^n$ telles que $C(x) = 1$?



Théorème 1 **CIRCUIT-SAT** est NP-complet.

Démonstration. D'abord **CIRCUITSAT** est dans NP :

Montrons que **CIRCUITSAT** soit NP-difficile. Soit **A** un problème dans NP et donnons une réduction de **A** dans **SAT**.

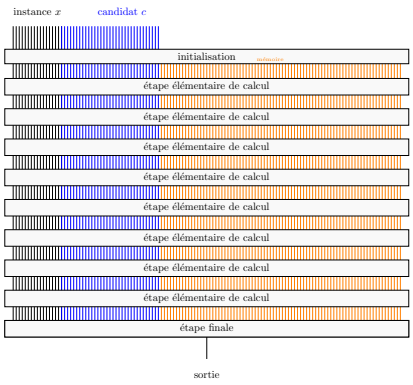


Comme **A** est dans NP, il existe un vérificateur V, il existe deux polynômes p, q tel que :

— pour toute instance x,
x est positive pour **A** ssi **il existe un mot** c de longueur $q(|x|)$ tel que $V(x, c)$ répond oui

— pour toute instance x, pour tout mot $c \in \{0,1\}^{q(|x|)}$, le temps d'exécution de $V(x, c)$ est majoré par $p(|x|)$.

Considérons une instance x de taille n. On peut construire en temps polynomial un circuit C_n qui prend en entrée $n + q(n)$ bits et qui simule n'importe quelle exécution de $V(x, c)$, où x est une suite de n bits et c une suite de $q(n)$:

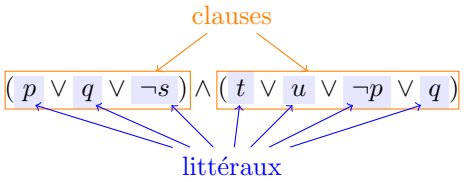


La réduction est :

fonction $tr(x)$
    $n = |x|$
    construire le circuit C_n
    renvoyer le circuit C_n sur lequel on a fixé les n premiers bits à $x_1, \ldots, x_n$

■

SAT
entrée : Une formule φ de la logique propositionnelle en forme normale conjonctive ;
sortie : oui si φ est satisfaisable ; non sinon.



Théorème 2 (de Cook) **SAT** est NP-complet.

Démonstration. On réduit **CIRCUITSAT** à **SAT** de la façon suivante. Pour valeur d du circuit, on note on note $in_1(d)$ et $in_2(d)$ les valeurs données en entrée de la porte. Considérons un circuit C.
On construit la formule :

$$fonctionnement(C) := \bigwedge_{d \text{ porte et}} (in_1(d) \wedge in_2(d)) \to d \wedge (d \to in_1(d)) \wedge (d \to in_2(d)) \\ \bigwedge_{d \text{ porte non}} (in_1(d) \vee d) \wedge (\neg in_1(d) \vee \neg d)$$

On pose $tr(C) = fonctionnement(C) \wedge d_{sortie}$. ■

2 3SAT

Définition 3 (3-forme normale conjonctive) Une 3-forme normale conjonctive est une formule normale conjonctive dans laquelle il y a au plus 3 littéraux dans une clause.

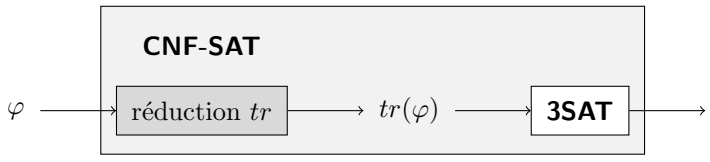
Exemple 4 $(p \vee q) \wedge (r \vee \neg p \vee q) \wedge (\neg r \vee s \vee \neg p)$.

Définition 5 **3-SAT**
entrée : Une formule φ de la logique propositionnelle en 3-forme normale conjonctive ;
sortie : oui si φ est satisfiable ; non sinon.

Remarque 6 On peut toujours supposer qu'il y a exactement 3 littéraux par clause en dupliquant des littéraux.

Proposition 7 **3-SAT** est NP-dur.

Démonstration.
Nous allons réduire polynomialement **CNF-SAT** à **3SAT**.



Si φ est une forme normale conjonctive, $tr(\varphi)$ est obtenue à partir de φ en remplaçant chaque clause par un ensemble de 3-clauses en introduisant des variables supplémentaires.

Exemple 8 $(a \vee b \vee c \vee d \vee e) \quad \rightsquigarrow \quad (a \vee b \vee \alpha) \wedge (\neg\alpha \vee c \vee \beta) \wedge (\neg\beta \vee d \vee e)$.

Voici l'algorithme pour tr :

entrée : une forme normale conjonctive φ
sortie : une 3-forme normale conjonctive φ' telle que φ satisfiable ssi φ' satisfiable
fonction $tr(\varphi)$
  $\varphi' := \top$
  φ s'écrit sous la forme $\bigwedge_{i=1..n} \bigvee_{j=1..n_i} \ell_{ij}$ où ℓ_{ij} sont des littéraux
  pour $i := 1..n$
    si $n_i = 1$ **alors**
      ajouter la clause ℓ_{i1} à φ'
    sinon
      ajouter la clause $\ell_{i1} \vee \alpha_{i2}$ à φ'
      pour $j := 2..n_i - 1$
        ajouter la clause $\neg\alpha_{i,j} \vee \ell_{ij} \vee \alpha_{i,j+1}$ à φ'
      ajouter la clause $\neg\alpha_{i,n_i} \vee \ell_{in_i}$ à φ'
  renvoyer φ'

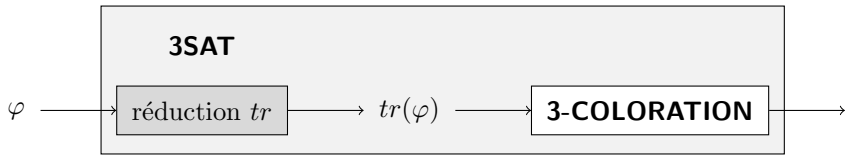
1. φ est une instance positive de **SAT** ssi $tr(\varphi)$ est une instance positive de **3SAT**.
   $\boxed{\Leftarrow}$ Supposons qu'il existe une valuation ν qui rende φ vraie (on écrit $\nu \models \varphi$ pour dire que cela). En particulier, chaque clause $\bigvee_{j=1..n_i} \ell_{ij}$ est vraie. Il existe donc $j \in \{1, \dots, n_i\}$ tel que $\nu \models \ell_{ij}$. On modifie ν en disant que $\alpha_{ij'}$ pour $j' < j$ est vraie, puis que $\alpha_{ij'}$ pour $j' \geq j$. Ainsi, toutes les clauses de φ' sont rendues vraies par ν.
   $\boxed{\Rightarrow}$ Réciproquement, supposons que $tr(\varphi)$ soit vraie pour une certaine valuation ν. Montrons $\nu \models \varphi$. Par l'absurde, supposons qu'une clause $\bigvee_{j=1..n_i} \ell_{ij}$ soit fausse pour ν. On aurait alors ν qui rend vraie $\alpha_{i2}, \dots, \alpha_{in_i}$ mais aussi qui rend faut α_{in_i}. Contraduction. On a bien $\nu \models \varphi$.
2. $tr(\varphi)$ est calculable en temps polynomial en la taille de φ.

■

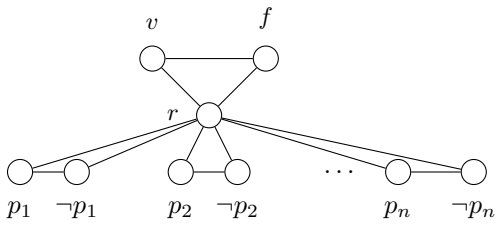
3 3COL

Théorème 9 **3-COLORATION** est NP-dur.

Démonstration.



On définit une réduction tr en temps polynomial qui à toute **3SAT**-instance φ associe une **3-COLORATION**-instance $tr(\varphi)$. $tr(\varphi)$ est un graphe basé sur le *gadget* suivant :



où $p_1, p_2, \dots, p_n$ sont les propositions atomiques qui apparaissent dans φ. On code le fait qu'un littéral est vrai par le fait qu'il a la couleur du sommet v et faux s'il a la couleur du sommet f. La couleur de $\neg x$ est toujours différente de celle de x.

Ensuite, pour chaque 3-clause, disons la clause numéro i de la forme $(\ell_{i1} \vee \ell_{i2} \vee \ell_{i3})$ de φ, on ajoute le gadget ci-dessous. Pour coder une 2-clause i de la forme $(\ell_{i1} \vee \ell_{i2})$, on prend le même gadget mais avec ℓ_{i2} à la place de ℓ_{i3}. La fonction tr est calculable en temps polynomial.

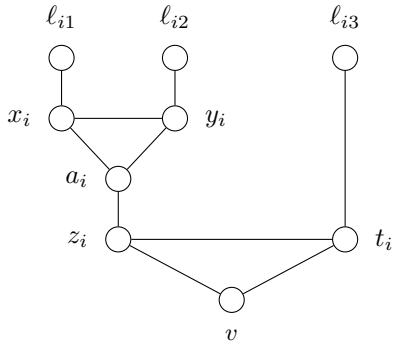
Formellement, pour $\varphi := \bigwedge_{i=1}^{m} \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$, on pose $tr(\varphi) = (V, E)$ avec

— $V := \{v, f, r, p_1, \neg p_1, \dots, p_n, \neg p_n\} \cup \bigcup_{i=1}^{m} \{x_i, y_i, a_i, z_i, t_i\}$;

—

$$\begin{aligned} E := &\{(v,f),(f,r),(r,v)\} \cup \\ &\bigcup_{j=1}^{n} \{(p_i, r), (\neg p_i, r)\} \cup \\ &\bigcup_{j=1}^{m} \{(\ell_{i1}, x_i), (\ell_{i2}, y_i), (\ell_{i3}, t_i), (x_i, y_i, (x_i, a_i), (y_i, a_i), (a_i, z_i), (z_i, t_i), (z_i, v), (t_i, v))\} \end{aligned}$$

Lemme 10 Dans toute coloration, l'un des sommets $\ell_{i1}, \ell_{i2}, \ell_{i3}$ a la couleur de v.



Démonstration. Par l'absurde, supposons qu'aucun n'est de la couleur de v. Alors ils sont tous de la couleur de f (on rappelle qu'ils ne peuvent pas avoir la couleur de r car la structure initiale ne le permet pas). Dans ce cas, comme ℓ_{i3} est de la couleur de f, pour sûr t_i est de la couleur de r donc z_i est de la couleur de f. D'autre part, comme ℓ_{i1} et ℓ_{i2} sont de la couleur de f, pour sûr, x_i et y_i ne sont pas de couleur de f donc a_i est de couleur de f. Contradiction. ■

Lemme 11 On peut compléter toute coloration où les sommets $\ell_{i1}, \ell_{i2}, \ell_{i3}$ sont de la couleur de f ou v et l'un d'eux est de la couleur de v.

Démonstration. Faire tous les cas. ■

Lemme 12 φ satisfiable ssi $tr(\varphi)$ est 3-coloriable.

Démonstration. $\boxed{\Rightarrow}$ Supposons que φ est satisfiable et soit ν une valuation telle que $\nu \models \varphi$. On colorie les noeuds de la façon suivante :

— $c[r] = 2$; $c[v\,] = 1$; $c[f\,] = 0$;
— $c[p] = \nu[p]$;
— $c[\neg p] = 1 - \nu[p]$.

Par le lemme 11, on complète la coloriation pour tous les gadgets. Donc $tr(\varphi)$ est 3-coloriable.

$\boxed{\Leftarrow}$ Supposons que $tr(\varphi)$ est 3-coloriable. On construit la valuation ν :

— $\nu[p_i] = 1$ si p_i est colorié avec la couleur de v : 0 sinon.

Par le lemme 10, $\nu \models \varphi$. ■ ■

4 Circuit hamiltonien dans un graphe orienté

5 Programmation linéaire entière

6 Arbres de décision

TODO: HR76.pdf

7 Idées

MAPF
Tetris
Super Mario Bros bien sûr

8 Notes bibliographiques

La classe NP a été définie initialement avec des machines de Turing non-déterministes par Cook en 1971 [Coo71]. Le non-déterminisme est aussi présent dans d'autres modèles de calcul : automates, automates à pile. Karp a donné et montré que plusieurs problèmes sont NP-complets [Kar72]. On trouve une grande liste de problèmes NP-complets dans [GJ79].

Références

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

Algorithme X

François Schwarzentruber

1 Motivation

Vous avez déjà joué au Sudoku. Vous essayez de mettre un 1, puis vous revenez sur votre choix. Bref, c'est du backtracking, un parcours en profondeur dans l'arbre des possibilités.

Le problème est qu'il faut des algos efficaces pour faire et défaire (dans le cas du Sudoku, écrire un 3 au Sudoku, puis l'effacer). Pour ça, on va utiliser des listes doublement chaînées, où c'est en $O(1)$ d'enlever et ajouter un maillon c :

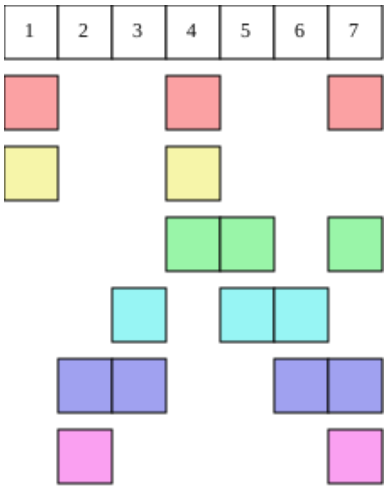
```
detach↔(c)
    c.→.← = c.←
    c.←.→ = c.→

attach↔(c)
    c.→.← = c
    c.←.→ = c
```

2 Définition de EXACT COVER

— Entrée : un univers $\mathcal{U}$ d'éléments, une collection $\mathcal{S}$ de sous-ensembles de $\mathcal{U}$;
— Sortie : $\mathcal{S}^* \subseteq \mathcal{S}$ tel que $\bigsqcup_{S \in \mathcal{S}^*} S = \mathcal{U}$, si un tel $\mathcal{S}^*$ existe ; impossible sinon.

Exemple 1



2.1 Motivation

Réduction de pavage d'échequiers avec de pentominos vers EXACT COVER :
— $\mathcal{U}$:= les cases de l'échequiers.
— Les sous-ensembles sont les cases couvertes par un pentominos à une certaine position

2.2 NP-complétude

EXACT COVER (problème de décision)
— Entrée : un univers $\mathcal{U} = \{1, \ldots, n\}$ d'éléments, une collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ avec $S_1, \ldots, S_m \subseteq \mathcal{U}$.
— Sortie : oui s'il existe $J \subseteq \{1, \ldots, m\}$ tel que $\bigsqcup_{j \in J} S_j = U$, non sinon.

Théorème 2 EXACT COVER est NP-complet.

Démonstration. Laissé en exercice. ■

3 Problème abstrait matriciel

— entrée : une matrice M de 0/1 de taille $n \times m$;
— sortie : $I \subseteq \{1, \ldots, n\}$ tel que il y a exactement un 1 dans chaque colonne de $M_{I,\cdot}$.

4 Algorithme de backtracking

entrée : une matrice de M
sortie : ensemble I d'indices de ligne de M tel que il y a exactement un 1 dans chaque colonne de $M_{I,\cdot}$, ou impossible si pas de tel I

```
fonction bt(M)
    si M n'a pas de colonnes alors
        renvoyer []
    sinon
        c := une colonne de M avec le moins de 1 dedans
        pour toutes les lignes r tel que M_rc = 1
            M.sélectionner la ligne r
            res = bt(M)
            si res ≠ impossible alors renvoyer r :: res
            M.désélectionner la ligne r
        renvoyer impossible
```

```
fonction M.sélectionner ligne r
    pour toutes les colonnes j tel que M_rj = 1
        M.couvrir colonne j
```

```
fonction M.couvrir colonne j
    M.supprimer colonne j
    pour toutes les lignes i tel que M_ij = 1
        M.supprimer ligne i
```

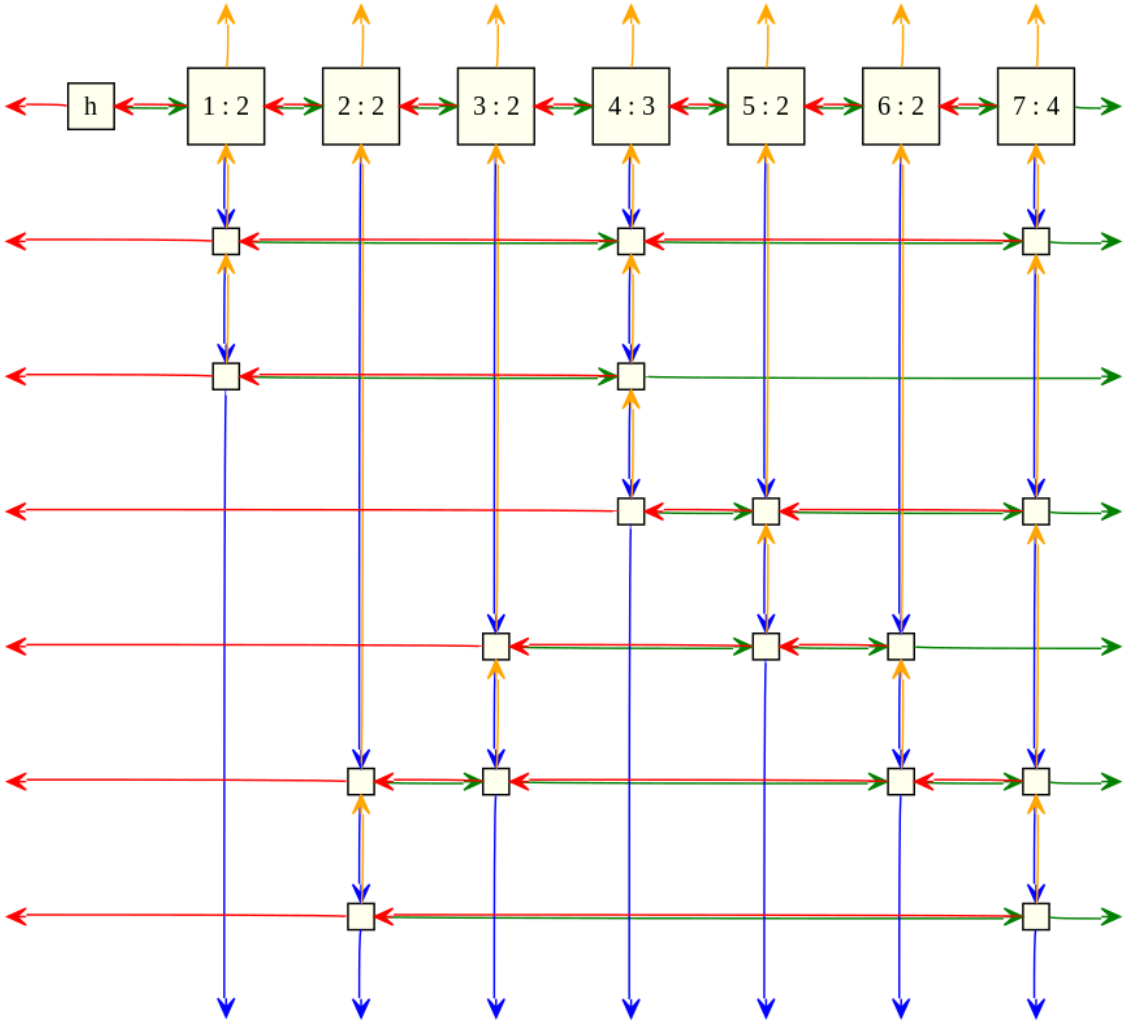
```
fonction M.désélectionner ligne r
    pour toutes les anciennes colonnes j tel que M_rj = 1
        M.découvrir colonne j
```

```
fonction M.découvrir colonne j
    pour toutes les anciennes lignes i / M_ij = 1
        M.remettre la ligne i
    M.remettre la colonne j
```

On aime une structure de données en place pour M où on peut sélectionner/désélectionner la ligne r.

5 Liens dansants

5.1 Description



On a une structure avec des listes doublement chaînées. On a nœud header M, des nœuds pour chaque colonne et des nœuds internes pour les 1 de la matrice. La structure ressemble à la matrice. l Dans la structure il y a un nœud header M. Des nœuds internes x (les petits carrés) avec les champs :

$$x.\leftarrow,\ x.\uparrow,\ x.\downarrow,\ x.\rightarrow,\ x.columnNode$$

Des nœuds c pour les colonnes, où il y a les même champs et en plus $y.\#$ qui indique le nombre de lignes i tel que $M_{ic} = 1$.

5.2 Faire des boucles

Dans les algorithmes qui suivent, on va parcourir les listes chaînées. Par exemple

pour toutes les lignes r tel que $M_{rc} = 1$

revient à considérer pour r les nœuds $c.\downarrow$, $c.\downarrow.\downarrow$, tant que $r \neq c$.
On peut écrire cela en C avec une boucle for :

```
for(r = c.down; r != c; r = r.down) {
    ...
    ...
}
```

ou alors une bouche while :

```
r = r.down
while(r != c) {
     ...
     ...
     r = r.down;
}
```

Ici, afin d'éviter de la lourdeur administrative, on note les parcours comme cela :
pour r parcourant$\downarrow$ c
  ...
pour r parcourant$\uparrow$ c
  ...
pour j parcourant$\rightarrow$ r
  ...
pour j parcourant$\leftarrow$ r
  ...

5.3 Fonction générale de backtracking

```
input: a 0/1 matrix represented by dancing links
output: a solution of the form a list of internal nodes, or impossible if there is no solution
bt(M):
       if(M.-> == M)
            return []
       else
            c = column node with c.# minimum

            for r traversing↓ c
                    M.select(r)

                    res = bt(M)
                    if res != impossible
                               return r :: res

                    M.unselect(r)
            return impossible

M.select(r):
     for j traversing→ r
          cover(j.columnNode)

M.unselect(r):
    for j traversing← r
         uncover(j.columnNode)
```

5.4 Trouver une colonne avec le moins de 1

fonction une colonne de M avec le moins de 1 dedans
  $m := +\infty$
  pour j parcourant$\rightarrow$ M
    si $j.\# < m$
      $j := c$
      $m = j.\#$
  renvoyer m

5.5 Couvrir/découvrir

Voici le pseudo-code qui couvre l'élément c. Ici, c désigne un nœud colonne. On commence par le détacher de la liste circulaire des colonnes. Vu que c est couvert, les sous-ensembles i contenant c ne sont plus disponibles. Il faut donc les supprimer. Pour supprimer le sous-ensemble i, on parcourt tous les éléments $j \neq c$ (nœuds internes) contenus dans i. Chacun de ces éléments j ne peuvent plus être couverts par i. On détache donc le nœud j. On décrémente aussi le nombre de sous-ensemble disponibles pour l'élément $j.columnNode$ car i n'est plus sélectionnable ; ça fait un de moins.

```
input: a column node c
effect: makes that the element c is "covered"
cover(c)
         detach↔→(c)
         for i traversing↓ c
                for j traversing→ i
                        detach↑↓(j)
                        decrement j.columnNode.#
```

Pour découvrir, il faut juste faire l'inverse.

```
input: a column node c
effect: makes that the element c is uncovered
uncover(c)
           for i traversing↑ c
                 for j traversing← i
                       increment j.columnNode.#
                       attach↑↓(j)
           attach↔→(c)
```

5.6 Attacher/détacher

On l'a déjà vu en début de cours. C'est comme ça :

```
detach↔→(c)
          c.→.← = c.←
          c.←.→ = c.→

attach↔→(c)
      c.→.← = c
      c.←.→ = c

detach↑↓(c)
     c.↑.↓ = c.↑
    c.↓.↑ = c.↑

attach↑↓(c)
    c.↑.↓ = c
    c.↓.↑ = c
```

Références