Fiche des principaux algorithmes et notions vus en MP2I/MPI

Ce document recense les principaux algorithmes au programme et quelques exemples pertinents des deux années de prépa MP2I/MPI.
Qui sommes nous ? Martin WATTEL et Balthazar RICHARD, deux 5/2 de la promo 2024-2025 de la MPI du lycée PAUL VALÉRY à Paris.
Bonne lecture, bonnes révisions et bon courage pour les concours ;) !

Sommaire

1 Algorithmes de tris

1.1 Tri rapide

Principe

Cet algorithme repose sur la recherche d'un pivot p sur lequel on va se baser pour séparer notre tableau en deux :

- à gauche les éléments inférieurs à p,
- à droite ceux supérieurs.

On applique ensuite récursivement l'algorithme sur chacune des deux parties du tableau jusqu'à ce qu'il soit trié.

Pseudo-code

```
entrée : tableau T, indice du premier élément f, indice du
         dernier élément l, indice du pivot p

partitionner (T, f, l, p) :
  T[p] <-> T[l]
  j <- f
  pour i allant de f à l-1
    si T[i] <= T[l]
      T[i] <-> T[j]
      j <- j+1
  T[l] <-> T[j]
  retourner j

tri_rapide (T, f, l) :
  si f < l
    p <- choix_pivot(T, f, l)
    p = partitionner(T, f, l, p)
    tri_rapide(T, f, p-1)
    tri_rapide(T, p+1, l)
```

En ce qui concerne le choix du pivot, on peut le prendre arbitrairement, aléatoirement ou bien prendre la médiane du sous tableau.

Complexité

`partitionner` est toujours en $\mathcal{O}(n)$. En revanche, le nombre d'appels récursifs à `tri_rapide` varie et on a :

dans le pire des cas : $\mathcal{O}(n^2)$.
dans le cas moyen : $\mathcal{O}(n\log(n))$.
dans le meilleur des cas : $\mathcal{O}(n\log(n))$.

Voir preuve

1.2 Tri par tas

Principe

On représente dans cet algorithme notre tableau sous forme d'arbre binaire et on cherche à obtenir un tas-min par le biais de percolations (ou tamisages) successives.

Voir la section tas binaires

Pseudo-code

```
entrée : arbre A, nœud N, taille n, longueur l

percoler (A, N, n) :
  k <- N
  j <- 2k
  tant que j <= n
    si j<n et A[j] < A[j+1]
      j <- j+1
    si A[k] < A[j]
      A[k] <-> A[j]
      k <- j
      j <- 2k
    sinon
      j <- n+1


tri_par_tas (A, l) :
  pour i allant de l/2 à 1
    percoler(A, i, l)
  pour i allant de l à 2
    A[i] <-> A[1]
    percoler(A, 1, i-1)
```

Complexité

dans le pire des cas : $\mathcal{O}(n\log(n))$.
dans le cas moyen : $\mathcal{O}(n\log(n))$.
dans le meilleur des cas : $\mathcal{O}(n\log(n))$.

1.3 Tri à bulle

Principe

On compare chaque élément aux autres afin de les ranger un à un en permutant deux éléments s'il le faut. On remarque que cet algorithme commence par trier les éléments les plus grands.

Pseudo-code

```
entrée : tableau T, taille n

pour i allant de n-1 à 1
  pour j allant de 0 à i-1
    si T[j+1] < T[j]
      T[j] <-> T[j+1]
```

Complexité

dans le pire des cas : $\mathcal{O}(n^2)$.
dans le cas moyen : $\mathcal{O}(n^2)$.
dans le meilleur des cas : $\mathcal{O}(n)$.

1.4 Tri fusion

Principe

On trie récursivement des sous tableaux puis on les fusionne.

Pseudo-code

```
entrée : A et B deux tableaux triées de tailles
         respectives a et b

fusion(A[1, ..., a], B[1, ..., b]) :
  si A vide
    renvoyer B
  si B vide
    renvoyer A
  si A[1] <= B[1]
    renvoyer A[1] @ fusion(A[2, ..., a], B[1, ..., b])
  sinon
    renvoyer B[1] @ fusion(A[1, ..., a], B[2, ..., b])


entrée : un tableau T de taille n.

tri_fusion(T[1, ..., n], n)
  si n <= 1
    renvoyer T
  sinon
    renvoyer fusion(tri_fusion(T[1, ..., ⌊n/2⌋]),
                    tri_fusion(T[⌊n/2⌋+1, ..., n]))
```

Complexité

`fusion` est en $\mathcal{O}(a+b)$ et `tri_fusion` est en $\mathcal{O}(n\log(n))$.

2 Algortihmes sur les graphes

2.1 Parcours en profondeur

Principe

On parcours un graphe $G = (S, A)$ en allant le plus loin possible de sommets en sommets.

Pseudo-code

```
entrée : graphe G, sommet s de G

p <- pile
vus <- tableau de faux de taille |G|
empiler(p, s)
tant que p non vide
  x <- dépiler(p)
  si x non vu :
    vus[x] <- vrai
    pour y voisin de x
      faire quelque chose
      empiler(p, y)
```

Complexité

La complexité du **parcours en profondeur** est en $\mathcal{O}(|S| + |A|)$.

2.2 Parcours en largeur

Principe

On parcours un graphe $G = (S, A)$ en visitant à chaque itération les sommets proches à une distance donnée de l'origine.

Pseudo-code

```
entrée : graphe G, sommet s de G

f <- file
vus <- tableau de faux de taille |G|
enfiler(f, s)
tant que f non vide
  x <- défiler(f)
  si x non vu :
    vus[x] <- vrai
    pour y voisin de x
      faire quelque chose
      enfiler(f, y)
```

Complexité

La complexité du **parcours en largeur** est en $\mathcal{O}(|S| + |A|)$.

2.3 Tri topologique

Principe

Le principe du **tri topologique** est de nous renvoyer une liste triée des sommets d'un graphe acyclique orienté $G = (S, A)$ dans laquelle s est avant t pour tout $(s, t) \in A$.

Attention : il n'y a pas nécessairement unicité du **tri topologique** car si u et v ne sont pas liés dans G, alors les ordres $(s_1, ..., u, ..., v, ..., s_n)$ et $(s_1, ..., v, ..., u, ..., s_n)$ sont les mêmes...

Pour en obtenir un, on réalise un parcours en profondeur d'un graphe G en notant chaque sommet dans l'ordre postfixe.

Rappel : préfixe, infixe puis postfixe (ou suffixe).

Pseudo-code

```
entrée : graphe G, sommet s de G

p1 <- pile
p2 <- pile
vus <- tableau de faux de taille |G|
empiler(p1, s)
tant que p1 non vide
  x <- dépiler(p1)
  si x non vu :
    vus[x] <- vrai
    pour y voisin de x
      empiler(p1, y)
    empiler(p2, x)
tant que p2 non vide
  dépiler(p2)
```

On obtient le **tri topologique** lorsque l'on dépile p2.

Complexité

La complexité du **tri topologique** est la même que celle d'un parcours en profondeur, soit en $\mathcal{O}(|S| + |A|)$.

2.4 Algorithme de Floyd-Warshall

Principe

L'algorithme de **Floyd-Warshall** permet de déterminer les distances des plus courts chemins entre toutes les paires de sommets dans un graphe orienté et pondéré ne possédant pas de circuit de poids strictement négatif.

On va chercher à construire les matrices $M_1, \cdots, M_n$ où

$$\forall (k,i,j) \in [\![1;n]\!]^3 \quad [M_k]_{i,j} := \min([M_{k-1}]_{i,j}, [M_{k-1}]_{i,k} + [M_{k-1}]_{k,j})$$

en sachant que M_0 désigne la matrice d'adjacence de G donnant le poids des arcs et ∞ s'il n'en existe pas.

Ainsi, pour $(k,i,j) \in [\![1;n]\!]^3$ $[M_k]_{i,j}$ désigne le poids minimal d'un chemin liant i à j n'empruntant que des sommets intermédiaires dans $\{1,...,k\}$.

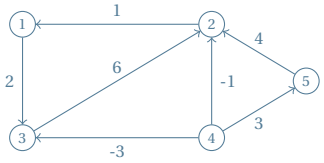
Pseudo-code

```
entrée : M_0 la matrice d'adjacence du graphe G = ({1,...,n}, A)

pour k allant de 1 à n
  pour i allant de 1 à n
    pour j allant de 1 à n
      [M_k]_(i,j) <- min([M_k-1]_(i,j), [M_k-1]_(i,k) +
                                        [M_k-1]_(k,j))
```

Un exemple

Prenons le graphe suivant, de matrice d'adjacence M_0 :



$$M_0 = \begin{pmatrix} 0 & \infty & 2 & \infty & \infty \\ 1 & 0 & \infty & \infty & \infty \\ \infty & 6 & 0 & \infty & \infty \\ \infty & -1 & -3 & 0 & 3 \\ \infty & 4 & \infty & \infty & 0 \end{pmatrix}$$

L'execution de l'algorithme de **Floyd-Warshall** nous donne :

$$M_5 = \begin{pmatrix} 0 & 8 & 2 & \infty & \infty \\ 1 & 0 & 3 & \infty & \infty \\ 7 & 6 & 0 & \infty & \infty \\ 0 & -1 & -3 & 0 & 3 \\ 5 & 4 & 7 & \infty & 0 \end{pmatrix}$$

Complexité

La complexité de l'algorithme de **Floyd-Warshall** est en $\mathcal{O}(n^3)$ où $n = |S|$.

2.5 Algorithme de Dijkstra

Principe

On considère un graphe $G = (S, A)$ pondéré et non orienté ainsi que s et t deux sommets dont on cherche le plus court chemin les reliant. Le principe va être de calculer étape par étape le meilleur chemin.

Algorithme

On va créer un tableau de $|S|$ colonnes et $|S|$ lignes et y calculer à chaque étape le meilleur chemin en sommant le poids des arêtes déjà parcourues.

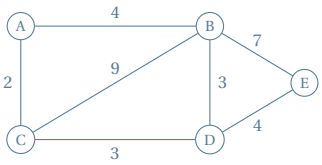
Pseudo-code

```
entrée : graphe pondéré G=(S,A,p),
         deux sommets s et t

file de priorité F <- {s}
x <- defile F
tant que x ≠ t
  pour chaque voisin y de x non vu
    si y n'est pas dans F
      ajouter y à F avec le poids d[x] + p(x,y)
    sinon et si d[x]+p(x,y) < F[y]
      modifier F[y]
renvoyer d[t]
```

Un exemple

Prenons le graphe suivant :



On crée le tableau des arêtes :

A	B	C	D	E	étapes
0	4-A	2-A			1
×	11-C	2-A	5-C		2
×	8-D	×	5-C	9-D	3
×	8-D	×	×	15-E	4
×	×	×	×	9-D	5

On obtient donc le chemin suivant : A→C→D→E d'une distance totale de 9.

Complexité

La complexité de l'algorithme de **Dijkstra** est en $\mathcal{O}(|S|^2 + |A|)$ pour une file quelconque et en $\mathcal{O}((|A| + |S|) \cdot \log(|S|))$ pour un tas, à condition que $|A| \ll |S|^2$.

2.6 Algorithme A*

Principe

Cet algorithme nous donne le plus court chemin entre deux sommets s et t d'un graphe pondéré $G = (S, A, p)$, en exploitant les connaissances que l'on possède pour trouver une solution le plus rapidement possible.

On utilise pour cela une heuristique h qui associe à chaque sommet une valeur, que l'on veut croissante avec la distance à la destination.
Par exemple pour se rendre d'une ville à une autre, on peut considérer la distance à vol d'oiseau.

Pseudo-code

```
entrée : G=(S,A,p), s, t, h

d <- tableau des distances (initialisé à ∞)
F <- file de priorité
(x, y) <- (s, h(s))
tant que x != t
  d[x] <- y - h(x)
  pour tout voisin z de x
    w <- d[x] + p(x, z) + h(z)
    si z présent dans F
      si la valeur associée est supérieur à w
        la remplacer par w
    sinon et si d[x] + p(x, z) < d[z]
      insérer (z, w) dans F
  (x, y) <- extraire(F)
renvoyer y
```

Complexité

La complexité dépend de l'heuristique, par son temps de calcul et par sa qualité.
L'algortihme A* est parfois plus lent que celui de Dijkstra : $\mathcal{O}(|S|^2 + |A|)$.

Propositions

h est **admissible** si elle ne surestime jamais la distance réelle à la destination :

$$\forall s \in S \quad h(s) \leq d(s,t).$$

Si l'heuristique est admissible, le chemin trouvé par A* est optimal.

h est **monotone** si :

$$\forall (x,y) \in S^2 \quad |h(x) - h(y)| \leq d(x,y).$$

Si h est monotone, A* ne traite jamais deux fois un même sommet.

2.7 Algorithme de Kruskal

Principe

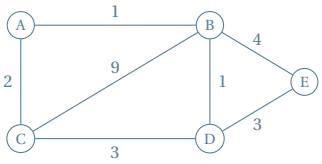
On considère un graphe $G = (S, A)$ pondéré dont on cherche un arbre couvrant minium.

Algorithme

- on trie les arêtes selon leurs poids,
- on ajoute les arêtes sur le graphe $G' = (S, \emptyset)$ par ordre croissant tant quelles ne rendent pas le graphe cyclique.

Un exemple

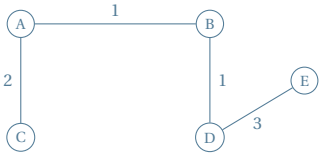
Prenons le graphe suivant :



On crée le tableau des arêtes :

AB	1
BD	1
AC	2
CD	3
DE	3
BE	4
BC	9

On obtient donc l'arbre couvrant minimal suivant :



sans avoir ajouté les arêtes CD, CB et BE.

Complexité

La complexité de l'algorithme de **Kruskal** est en $\mathcal{O}(|A| \cdot (|S| + \log(|A|)))$ sans utiliser une structure de type unir et trouver, et en $\mathcal{O}(|A| \cdot \log(|S|))$ avec.

2.8 Algorithme de Kosaraju

Principe

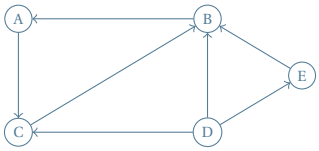
On considère un graphe orienté $G = (S, A)$ dont on cherche à déterminer les composantes fortement connexes.

Algorithme

- on réalise un parcours en profondeur sur G et on note l'ordre suffixe,
- on réalise un parcours en profondeur sur $G^\top$, le graphe transposé de G, selon l'ordre précédent.

Un exemple

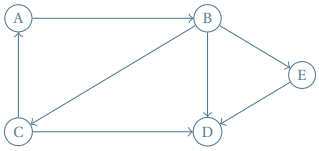
Prenons le graphe suivant :



Un premier parcours en profondeur nous donne l'ordre suivant :

B−C−A E D

Le graphe transposé est le suivant :



Le parcours en profondeur selon l'ordre établie précédemment nous donne les CFC suivantes :

{A, B, C}, {E} et {D}

Complexité

La complexité de l'algorithme de **Kosaraju** est en $\mathcal{O}(|S| + |A|)$.

3 Algorithmes de jeux

3.1 Calcul des attracteurs

Principe

On étudie des **jeux d'accessibilité** à deux joueurs, ce qui peut être vu comme un graphe où chaque sommet est un état possible du jeu, les arcs sont les coups jouables. On parle d'arène lorsqu'on a un graphe $G = (S, A)$ et deux ensembles S_1 et S_2 qui partitionnent S. Lorsque l'on est sur un sommet S_i, c'est au joueur $i \in \{1,2\}$ de jouer. Une partie depuis s_0 est un chemin de s_0 à un état de fin de partie.
Les positions gagnantes pour le joueur i sont :

- les sommets sans successeurs assignés au joueur i,
- les sommets du joueur i ayant un successeur qui est une position gagnante,
- les sommets du joueur $i+1 \mod 2$ (l'adversaire) dont tous les successeurs sont des positions gagnantes pour i.

On définit la suite $(\text{Attr}_k^i)_{k\in\mathbb{N}}$ d'**attracteurs** du joueur i récursivement :

$$\text{Attr}_0^i := \{s \in S_i \mid \text{d}^-(s) = 0\}$$
$$\text{Attr}_{k+1}^i := \text{Attr}_k^i \cup \{s \in S_i \mid \exists t \in \text{Attr}_k^i \wedge (s,t) \in A\} \cup \{s \in S_{i+1 \mod 2} \mid \forall (s,t) \in A \ \ t \in \text{Attr}_k^i\}$$

Algorithme

- on réalise un parcours en profondeur sur $G^\top$ depuis les sommets sans prédécesseurs,
- pour les sommets de S_i, on les ajoute et on fait un appel récursif sur ses voisins,
- pour les sommets de $S_{i+1 \mod 2}$, on veut le rajouter si tous ses prédécesseurs ont été ajoutés,
- on augmente $T[s]$ de 1 où T est un tableau indiquant le nombre de prédécesseurs ajoutés.

Pseudo-code

On propose l'algorithme naïf suivant

```
entrée : G=(S,A), S_i, S_{i+1 mod 2}, i

Attr <- {s∈S_i | d⁻(s)=0}
tant que Attr a été modifié
  mettre à jour Attr
renvoyer Attr
```

que l'on peut améliorer si G est acyclique ou bien en travaillant sur le graphe transposé (algorithme ci dessus) :

```
entrée : G=(S,A), S_i, S_{i+1 mod 2}, i

d <- tableau des degrés
T <- tableau indiquant le nombre de prédécesseurs  ajoutés
Attr <- tableau renvoyé
deja_traités <- tableau des sommets traités
G^⊤ <- graphe transposé
pour s∈S
  si d(s)=0
    parcours en profondeur dans G^⊤
renvoyer Attr
```

Complexité

La complexité du premier algorithme est en $\mathcal{O}(|S| \cdot (|A| + |S|))$ tandis que le second est en $\mathcal{O}(|A| + |S|)$.

3.2 Algorithme Min-Max

Principe

Cet algorithme permet d'étudier partiellement le graphe des coups possibles d'un jeu afin de déterminer le meilleur coup à jouer.

Notre heuristique assigne une plus grande valeur à une position favorable qu'à une position défavorable.

On borne la profondeur de l'arbre pour rester sur un nombre raisonnable de coups à considérer.

Pour trouver ce qui semble être le meilleur coup depuis la configuration c, il suffit de calculer argmax(Min-Max(c', h, i, p)) avec c' successeur de c,
argmax($f(i)$) étant le $i \in I$ tel que $f(i)$ soit maximal.

Pseudo-code

```
entrée : configuration c, heuristique h, joueur i, profondeur p

si p vaut 0
  renvoyer h(c)
sinon
  si c'est à i de joueur
    renvoyer max(Min-Max(c', h, i, p-1)) pour c' successeur de c
  sinon
    renvoyer min(Min-Max(c', h, i, p-1)) pour c' successeur de c
```

Complexité

La complexité de **Min-Max** dépend de l'heuristique et du jeu : plus il y a de coups possibles, plus ce sera lent.

3.3 Algorithme d'élagage $\alpha - \beta$

Principe

L'objectif est d'optimiser le calcul des différentes valeurs pour pouvoir augmenter la profondeur d'étude de l'algorithme Min-Max.

On explore l'arbre seulement partiellement sans perdre en qualité de l'étude.

Le temps d'execution va varier en fonction du nombre de fois où l'on élimines des branches.

Pseudo-code

```
entrée : configuration c,
         heuristique h,
         joueur i,
         pronfondeur p,
         int a, (a et b indiquent entres quelles valeurs on doit
         int b  chercher pour quelles influences le résultat final)

si p = 0
  renvoyer h(c)
si c'est au joueur i de jouer
  (on veut s'arrêter dès qu'on trouve une valeur supérieur à b)
  maxi = -∞
  pour c' successeur de c
    maxi = max(maxi, Elagage(c', h, i, p-1, maxi, b))
    si maxi > b
      renvoyer maxi
  renvoyer maxi
sinon
  (on veut s'arrêter dès qu'on trouve une valeur inférieur à a)
  mini = +∞
  pour c' successeur de c
    mini = min(mini, Elagage(c', h, i, p-1, a, mini))
    si mini < a
      renvoyer min
  renvoyer min
```

Complexité

La complexité de **l'élagage $\alpha - \beta$** dépend de l'ordre dans lequel on considère les successeurs. On peut ne pas avoir de chance et elaguer peu, mais aussi affiner l'heuristique pour avoir une meilleur analyse.

Si celle ci est trop complexe, cela peut limiter la profondeur à laquelle on peut explorer.

4 Théorie des automates

4.1 Déterminisation d'un automate

Principe

Le principe de l'algorithme est de transformer un automate non deterministe $A = (\Sigma, Q, I, F, \delta)$ en automate déterministe $A' = (\Sigma, Q', q_0, F', \delta')$.

Algorithme

Avec les mêmes notation que ci-dessus,

- $Q' := \mathscr{P}(Q)$
- $q_0 := \{I\} \in Q'$
- $F' := \{E \in \mathscr{P}(Q) \mid E \cap F \neq \emptyset\}$
- $\forall E \in Q' \quad \forall a \in \Sigma^* \quad \delta'(E, a) := \bigcup_{q \in E} \delta(q, a)$

Complexité

L'automate produit par ce procédé à un nombre d'états exponentiel en celui de l'automate initial.

4.2 Algorithme d'élimination des états

Principe

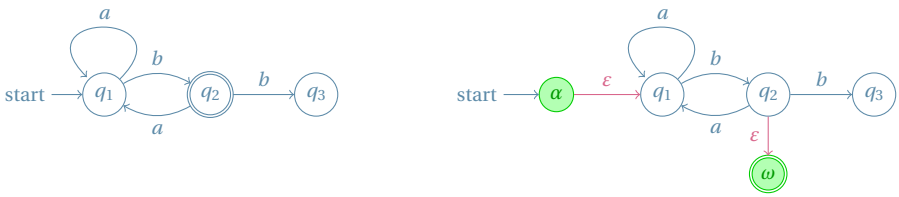
L'algorithme d'**élimination des états** consiste à determiner l'expression régulière associée à un graphe à partir de ce dernier.

Algorithme

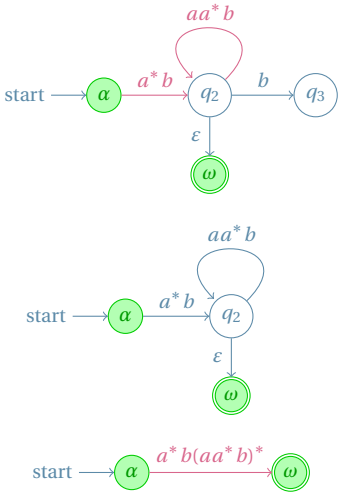
- ajouter α et ω deux nouveaux états : α le nouvel état initial, ω le nouvel état final,
- éliminer les états : on retire un sommet et on relie les arcs en conséquence, c'est-à-dire que si (s_1, s_r) et (s_r, s_2) lient respectivement q_1 à q_r et q_r à q_2 avec comme étiquettes e_1 et e_2, et que l'on retire q_r, alors on ajoute l'étiquette $e_1 e_2$ à l'arc liant s_1 à s_2 (que l'on crée si inexistant).

Un exemple

Considérons le graphe suivant auquel on ajoute les nouveaux états α et ω :



l'algorithme d'élimination des états s'execute comme suit :



Ainsi, l'automate du départ est caractérisé par l'expression $e = a^*b(aa^*b)^*$.

Complexité

L'algorithme d'**élimination des états** s'execute en $\mathcal{O}(|S|\cdot|A|)$.

4.3 Algorithme de Berry-Sethi

Principe

L'algorithme de **Berry-Sethi** ou **Construction de Glushkov** permet de construire un automate à partir d'une expression rationnelle.

Algorithme

- on commence par linéariser l'expression : chaque lettre ne doit apparaître qu'une seule fois donc on la numérote à chaque apparition,
- on détermine les ensembles $P(e')$, $D(e')$ et $F(e')$ où e' est l'expression linéarisée de e et

$$P(L) := \{e \in \Sigma \mid \exists m \in \Sigma^* \ni em \in L\}$$
$$D(L) := \{e \in \Sigma \mid \exists m \in \Sigma^* \ni me \in L\}$$
$$F(L) := \{u \in \Sigma^2 \mid \exists (m_1, m_2) \in (\Sigma^*)^2 \ni m_1 u m_2 \in L\}$$

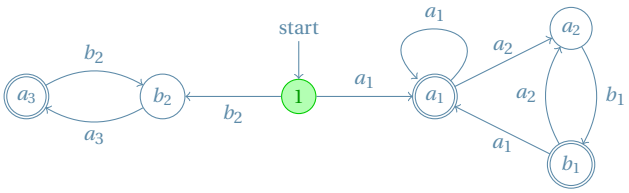
- on construit l'automate en conséquence :
  - on crée les sommets étiquetés par l'alphabet linéarisé plus un sommet initiale et on entoure les sommets de $D(e')$,
  - on lie le sommet initial 1 au sommets de $P(e')$,
  - on lie deux sommets a_i et b_j si $a_i b_j \in F(e')$,
- on retire les indices des lettres afin d'obtenir un automate reconnaissant e.

Un exemple

Considérons l'expression rationnelle $e = (a(ab)^*)^* + (ba)^*$.
On linéarise l'expression : $e' = (a_1(a_2b_1)^*)^* + (b_2a_3)^*$.
On en déduit

$$P(e') = \{a_1, b_2\}$$
$$D(e') = \{a_1, b_1, a_3\}$$
$$F(e') = \{a_1a_2, a_1a_1, a_2b_1, b_1a_1, b_1a_2, b_2a_3, a_3b_2\}$$

Ainsi, on obtient l'automate :



Complexité

La complexité de l'algorithme de **Berry-Sethi** est en $\mathcal{O}(n^2)$ où n est le nombre de caractères de e.

5 Algorithmes d'apprentissage

5.1 Algorithme des k plus proches voisins : algorithme naïf

Principe

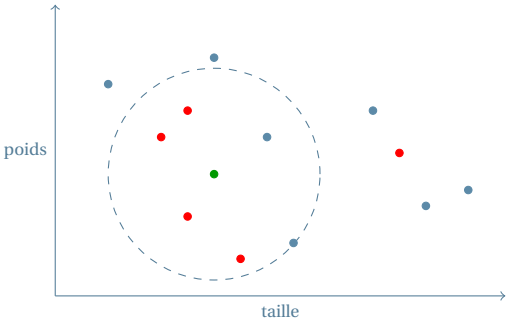
Étant donné un ensemble de classes et un individu que l'on cherche à ranger, on se donne une distance et on attribut à notre individu la classe majoritaire parmi ses k plus proches voisins.

Algorithme

- on calcule la distance de chaque point à notre représentant,
- on trie le tableau des distances par ordre croissant,
- on détermine la classe majoritaire parmi ces k éléments.

Un exemple

Considérons les points suivants, pouvant représenter des chiens et des chats placés selon leurs tailles et leurs poids, et donnons nous un individu dont on ne connait pas l'espèce.



Avec $k = 6$ et la distance euclidienne, les voisins sont deux chiens et quatre chats et donc notre individu est donc un chat selon l'algorithme.

Complexité

La complexité de l'algorithme des ***k* plus proches voisins** exécuté sur n points est en $\mathcal{O}(n\log(n))$.

Remarques

Si k est trop petit, on est sujet aux variations du jeu de donnée car les prédictions sont trop proche de celui-ci : l'algorithme fait du **sur-apprentissage**.

Au contraire, si k est trop grand, généralise trop pour répondre à un cas particulier : c'est du **sous-apprentissage**.

5.2 Algorithme des k plus proches voisins : Les arbres kd

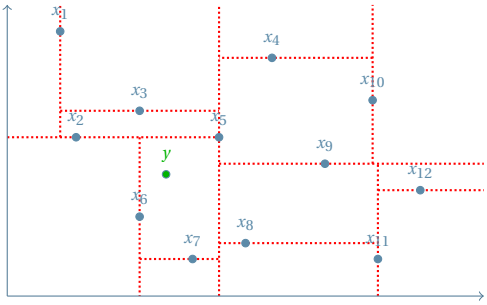
Principe

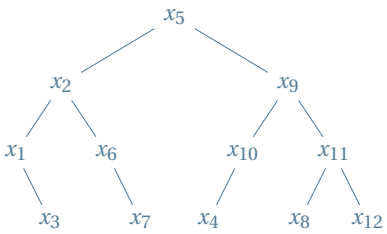
Cette méthode permet d'organiser le stockage des éléments du jeu d'entrainement afin de ne pas avoir à explorer tous les sommets mais plutôt les plus proches.

Algorithme

- on découpe l'espace par des segments verticaux et horizontaux (on alterne une fois sur deux, dans le cas de la dimension 2) passant par chaque point,
- on représente ce découpage à l'aide d'un kd arbre, (à gauche le sommet à gauche ou sous le sommet considéré, à droite le sommet à droite ou sous celui-ci),
- on parcours l'arbre en fonction de la position du sommet inconnu y (on se repère avec les coordonnées),
- les k derniers sommets sont les premiers candidats, que l'on insère dans une file de priorité organisée selon la distance à y,
- on remonte l'arbre en calculant à chaque nœud x_k la distance de y à l'hyperplan h_k associé au nœud :
  - si celle ci est inférieur à la distance $d(x_k, y)$, alors on parcours l'autre branche de x_k et on compare $d(x_k, y)$ et $d(x_{\min}, y)$ afin d'échanger $x_{\min}$ et y si le sommet est meilleur,
  - sinon, on continue à remonter l'arbre.

Un exemple





Pour $k = 2$:

- les deux premiers candidats sont $[x_6, x_7]$;
- on remonte en x_6 :
  déjà dans la file
  pas de sous-arbre gauche ;
- on remonte en x_2 :
  $d(x_7, y) < d(x_2, y) \rightarrow$ pas de modification de la file
  $d(x_7, y) > d(h_2, y) \rightarrow$ parcours du sous-arbre gauche en x_2 ;
- on descent en x_3 :
  $d(x_7, y) > d(x_3, y)$ et $d(x_6, y) < d(x_3, y) \rightarrow [x_6, x_3]$;
- on remonte en x_1 :
  $d(x_3, y) > d(x_1, y) \rightarrow$ pas de modification de la file
  pas de sous-arbre gauche ;
- on remonte en x_5 :
  $d(x_3, y) < d(x_5, y)$ et $d(x_5, y) < d(x_6, y) \rightarrow [x_6, x_5]$
  $d(x_5, y) < d(h_5, y) \rightarrow$ parcours du sous-arbre droit en x_5 ;
- on descent en x_8 :
  $d(x_5, y) < d(x_8, y) \rightarrow$ pas de modification de la file ;
- on remonte en x_{11} :
  $d(x_5, y) < d(x_{11}, y) \rightarrow$ pas de modification de la file
  $d(x_5, y) < d(h_{11}, y) \rightarrow$ pas de parcours ;
- on remonte en x_9 :
  $d(x_5, y) < d(x_9, y) \rightarrow$ pas de modification de la file
  $d(x_5, y) > d(h_9, y) \rightarrow$ parcours du sous-arbre gauche en x_9 ;
- on descend en x_4 :
  $d(x_5, y) < d(x_4, y) \rightarrow$ pas de modification de la file ;
- on remonte en x_{10} :
  $d(x_5, y) < d(x_{10}, y) \rightarrow$ pas de modification de la file
  pas de sous-arbre droit ;
- on remonte en x_9 : ...
- on remonte en x_5 : ...

Les deux plus proches voisins sont 6 puis 5.

Complexité

En moyenne, il faut explorer $\mathcal{O}(k \cdot \log(n))$ sommets pour trouver les k plus proches.

5.3 Algorithme des k-moyennes

Principe

Étant donné un ensemble de point dans un plan, on cherche à trouver des cluster (ou amas).

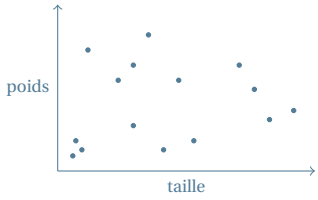
Algorithme

- on commence par prendre k représentants aléatoirement,
- on calcul ensuite les k clusters : un point appartient au cluster $i \in [\![1;k]\!]$ si sa distance à i est inférieure aux autres,
- on calcul les nouveaux représentants comme étant moyenne des points de leurs clusters.

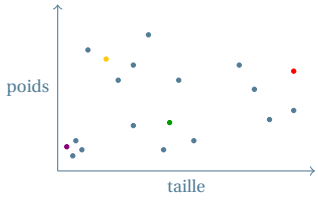
On recommence ensuite à l'étape 2 jusqu'à ce que cela nous convienne (clusters stables, critère de temps, d'itérations...)

Un exemple

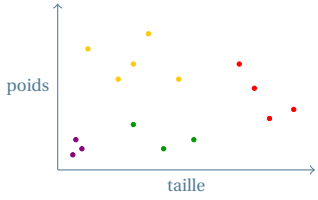
Considérons les points suivants, pouvant représenter des animaux par exemple, sur lesquels on va appliquer l'algorithme des 4-moyennes afin de les classifier selon l'espèce :



On places les 4 représentants suivants, aléatoirement :



On obtient donc les cluster suivants (on s'arrête par ailleurs ici) :



Complexité

La complexité de l'algorithme des **k-moyennes** exécuté sur n objets est en $\mathcal{O}(\alpha n)$ où α correspond au nombre d'itérations.

5.4 Algorithme ID3

Principe

Le but de l'algorithme **ID3** est de construire un arbre de décision, c'est-à-dire un arbre tel que chaque nœud est une question, chaque branche est une réponse possible et chaque feuille est une étiquette.

Définitions

On définie une **fonction d'entropie** H servant à calculer à quel point une question est pertinente. Pour un ensemble S donné, on le partitionne en $(C_1, ..., C_n)$, où pour $i \in [\![1; n]\!]$, C_i est l'ensemble des éléments d'étiquette i.

On pose ainsi, avec $n := \sum_{i=1}^{n} |C_i|$ et la convention que $0 \cdot \log(0) := 0$,

$$H(S) := -\sum_{i=1}^{n} \frac{|C_i|}{n} \log\left(\frac{|C_i|}{n}\right)$$

Notons que $H(S)$ est maximal lorsque un seul des $(|C_i|)_{1 \leq i \leq n}$ est non nul.

On pose aussi $S_{a=a_k} := \{x \in S \mid x_a = a_k\}$ et on définit le **gain d'entropie** d'une question sur la coordonnée a par

$$G(S, a) := H(S) - \sum_{k} \frac{|S_{a=a_k}|}{|S|} H(S_{a=a_k})$$

Pseudo-code

```
entrée : ensemble de données S
         ensemble de questions Q (souvent composé d'une question par coordonnée)

si Q = ∅
  renvoyer feuille(e) où e est l'étiquette la plus représenté de S
si S = ∅
  renvoyer feuille(e') ou e' est l'étiquette la plus représenté du
    parent
si H(S) = 0 (s'il y a une seule étiquette dans S)
  renvoyer feuille(e)
pour j allant de 0 à |Q|−1
  calculer Σ_{S_i} (S_i/S) H(S_i) pour la question q_j
on choisi j qui minimise la somme
on réalise un appel récursif sur chacun des S_i correspondant avec
    Q\{q_j} pour obtenir les fils f_1,...,f_i
on renvoie Nœud(q_j, f_1, ..., f_i)
```

Complexité

Si le nombre de coordonnées est très élevé, il est probable que l'une d'entre elle soit une très bonne question. Le temps d'execution va varier en fonction du nombre de fois où l'on élimines des branches.

5.5 Algorithme de classification hiérarchique ascendante

Principe

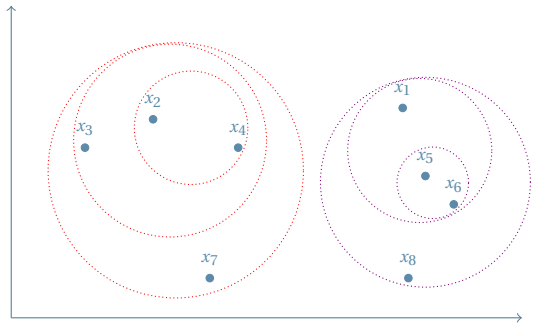
L'algorithme de **classification hiérarchique ascendante** (ou **CHA**) sert à la résolution de problèmes de clustering, c'est-à-dire à rassembler les éléments d'un ensemble S les plus proches localement afin de partionner S en k sous-ensembles.

Algorithme

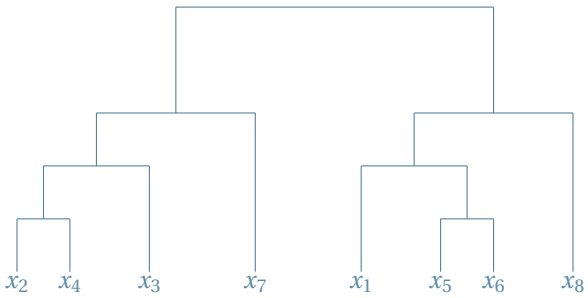
Il suffit de rassembler de manière gloutonne les points les plus proches, ce qui permet de dessiner un dendrogramme.

Un exemple

Considérons la distribution de points suivante et le paramètre $k = 2$,



on en déduit le dendrogramme suivant



Complexité

En moyenne, l'aglorithme de **classification hiérarchique ascendante** a une complexité en $\mathcal{O}(n^3 \cdot D)$, où D est le temps de calcul d'une distance.

6 Algorithmes du texte

6.1 Algorithme de Boyer-Moore

Principe

L'algorithme de **Boyer-Moore** est un algorithme de recherche de patterns dans un texte.

Pseudo-code

```
entrée: la chaîne de caractère c et le mot m à chercher dedans
sortie: l'indice de la première occurence de m dans c

offset <- taille de m - 1
tant que offset < taille de c faire
  i <- taille de m  - 1
  tant que m[i] = c[offset+i] faire
    i <- i - 1
  si i = -1 faire
    renvoyer offset
  offset <- offset + taille de m - i
renvoyer -1 (indique que le mot n'apparaît pas dans la chaîne de caractère)
```

ou en Ocaml

```
let boyer_moore (m: string) (c: string) : int =
  let taille_m = String.length m in
  let taille_c = String.length c in
  let rec parcours (offset: int) : int =
    if offset+taille_m <= taille_c then
      let i = ref (taille_m-1) in
      while !i >= 0 && m.[!i] = c.[!i+offset] do
        i := !i - 1
      done;
      if !i = -1 then
        offset
      else
        parcours (offset + taille_m - !i)
    else -1
  in parcours 0
```

Complexité

La complexité de l'algorithme de **Boyer-Moore** est en $\mathcal{O}(np)$ dans le pire des cas, avec n la taille de la chaîne et p la taille du mot m.

6.2 Algorithme de Rabin-Karp

Principe

L'algorithme de **Rabin-Karp** utilise la puissance des fonctions de hachage pour retrouver un motif dan une chaîne de caractère. On pose hach cette fonction de hachage qui prend une chaîne de caractère. On sait que si hach(portion$_1$) $\neq$ $hach$(portion$_2$), alors forcément les deux textes ne sont pas égaux, mais si hach(portion$_1$) = hach(portion$_2$) on ne sait pas si il sont égaux, il faudra dès lors tester caractère par caractère. Mais cet algorithme et cependant bien meilleur en terme de complexité si on a la fonction de hachage adaptée.

Voir la section table de hachage

Pseudo-code

```
entrée : un mot m, une chaîne de caractère c
sortie : l'indice de la première occurence de m dans c

h <- hach(m)
pour i allant de 0 à taille de c - taille de m - 1 faire
  si h = hach(c[i ... i + taille de c - 1] alors
    tester si c[i + taille de c - 1] = m, si oui retourner i
```

6.3 Algorithme LZW

Principe

L'algorithme **LZW** est un algorithme de compression de données sans perte inventé par LEMPEL-ZIV-WELCH.
Le but de l'algorithme est le suivant : réduire la taille du texte passé en entrée, en réduisant les motifs les plus utilisés, et sans perte, qui caractérise le fait qu'il est possible de reconstitué le texte une fois compresser dans une phase de compression.

Compression

Pour la phase de compression, on va utiliser un dictionnaire (qui va associé un motif à un code) qui sera initialement peuplé de tous les caractère (comme par exemple l'ASCII). Le but, peupler ce dictionnaire de motifs que l'on rencontre souvent, remplaçant ainsi les valeurs dans le texte par leur code. On considera que le code est incrémenté pour chaque nouvelle entrée dans le dictionnaire par rapport au plus grand code déjà présent dans celui-ci.

Un exemple

'a' a le code 97 en ASCII.

Le principe est le suivant, on va parcourir chaque lettre du texte tout en gardant en mémoire un motif tampon t. On va alors distinguer deux cas sur la lettre l rencontrée :

— Si $t \oplus l$ (on dénote $\oplus$ la concaténation de deux mots) est dans le dictionnaire, on remplace t par $t \oplus l$

— Sinon, on ajoute $t \oplus l$ dans le dictionnaire, et on renvoie le code de celui-ci pour le texte finale. On remplace dès lors t par la lettre l.

On propose le pseudo-code suivant :

```
1  entrée: m
2  sortie: m' le mot m compressé
3
4  t <- m[0]
5  D <- dictionnaire
6  m' <- "" (mot résultant)
7  pour i allant de 1 à taille de m - 1 faire
8      t' <- t ⊕ m[i]
9    si t' n'est pas dans D alors
10     ajouter t' dans D
11         m' <- m' ⊕ D[t']
12         t <- m[i]
13     Sinon:
14         t <- t ⊕ m[i]  (mot tampon à tester dans le langage L)
```

Décompression

La phase de décompression est similaire à la phase de compression, en un sens elle est l'inverse de celle-ci. On va également introduire un dictionnaire comme précedemment. Ici, on va raisonner sur les codes renvoyés à la suite de la compression. On va garder également en mémoire un tampon comme précédemment noté t qui va lui enregistrer les motifs visités.
Pour chaque code rencontré dans le texte, on va peupler t du motif du code dans D. Si, t n'apparaît pas dans D, on l'ajoute, et on réinitialise t au motif du code courant. À chaque itération, on va renvoyer le motif du code courant dans D. Or, il se peut que le code courant ne soit pas dans le dictionnaire D, on sait alors que (refaire sur un exemple simple comme "lalalalere") il s'agit du motif du code précedemment rencontré, et de la première lettre du motif associé au code le plus haut dans le dictionnaire D.

6.4 Algorithme de Huffman

Présentation

L'algorithme de compression de **Huffman** est, comme son nom l'indique, un algorithme de compression, permettant de réduire l'espace nécessaire pour coder un mot.
Il a été théorisé par David Albert Huffman en 1952.

Principe

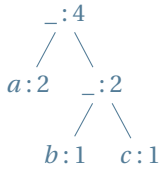
Pour un mot m, on va d'abord chercher à calculer les occurrences de chaque lettre dans celui-ci. Prenons par exemple : $m = aabaac$
On va avoir un tableau associant chaque occurrence des lettres de m :
ici : $[a:4, b:1, c:1]$
Les opération de l'algorithme sont les suivantes :

— on va chercher les deux mots avec l'occurrence la plus petite.
Par exemple, ici $b:1$ et $c:1$;

— on va les "fusionner", c'est-à-dire, créer un nœud qui a pour enfants les deux nœuds, et ayant également une valeur, représentant la somme des deux occurrences. Par la suite, on représentera les éléments du tableau comme des nœuds ;

— si il n'y a plus qu'un seul nœud, on s'arrête.

Un exemple



Ayant le tableau : on va pouvoir représenter les lettres de la manière suivante : les lettres étant des feuilles, on va chercher à accèder à la feuille en question, on va alors avoir un certains chemin de la racine à celle-ci. Pour représenter le chemin, on va utiliser la convention suivante : si on visite le fils gauche du tableau, on représente cette étape par un 0, et si on va à droite, on le représentera par un 1.

Ici, b a pour chemin de la racine à sa feuille : 10

Pour la compression, on va alors remplacer toutes les lettres du texte par le code du chemin précedemment explicité, et renvoyer avec le texte compressé l'arbre construit.

Pour la décompression, il suffit de parcourir tous les codes rencontrés dans le message jusque former le mot.
Par exemple, le texte $aabc$ va être transformer en 001011, il faudra simplement pour chaque bit parcourir l'arbre ci-dessus, jusque obtenir une feuille dans celui-ci, dès lors on sait que le caractère a apparaît au début avec 0, puis a également, puis b avec 10, et finalement c avec 11.

7 Algorithmes probabilistes

7.1 Algorithmes de Monte Carlo

Principe

L'idée des algorithmes de **Monte Carlo** est de réaliser de nombreuses expériences aléatoires dont on connaît le temps de calcul afin de s'approcher (on peut se tromper même si le but est de minimiser l'échec) du résultat à une question sans avoir à utiliser d'algorithme trop demandant en ressources.

On dit qu'on probabilise le résultat.

Un exemple

Il en existe plusieurs comme par exemple la détermination de l'air sous une courbe, de la surface d'un lac ou encore d'une approximation de la valeur du nombre π.

Considérons pour l'exemple trois matrices A, B et C de $M_n(\mathbb{R})$ dont nous voulons savoir si $AB = C$. Le calcul du produit AB (puis la comparaison) se fait en $\mathcal{O}(n^3)$ ou bien en $\mathcal{O}(n^\varepsilon)$ avec $\varepsilon \in]2;3[$ avec l'algorithme de Strassen.

Ici, on va employer un algorithme de **Monte Carlo** comme suit :
on crée un vecteur aléatoire $X \in M_{n,1}(\mathbb{Z}/2\mathbb{Z})$ puis on calcul $Y = BX$, $Z = AY$ et CX et on les compares en $\mathcal{O}(n^2)$.

Plus on augmente le nombre d'itérations, plus la précision augmente (pour 30 itérations, on a plus de chances de gagner l'euro million que de se tromper avec l'algorithme...)

Complexité

La complexité de ces algorithmes dépend de l'algorithme mais elle a le bon goût d'être connu à l'avance en sachant le coût d'une expérience aléatoire et le nombre d'itérations.

7.2 Algorithmes de Las Vegas

Principe

L'idée des algorithmes de **Las Vegas** est de réaliser autant d'expériences aléatoires qu'il faut pour trouver une solution correcte à un problème donné.

On dit qu'on probabilise le temps d'execution : l'algorithme renvoie toujours le bon résultat mais le temps d'execution suit une variable aléatoire.

Un exemple

Un exemple est le tri rapide dit randomisé, où la sélection du pivot se fait de manière aléatoire. Celui possède une complexité dans le pire des cas en $\mathcal{O}(n^2)$ et une complexité moyenne en $\mathcal{O}(n \cdot \log(n))$ (tout comme dans le meilleur des cas).

8 Synchronisation

8.1 Algorithme de Peterson

Principe

L'algorithme de **Peterson** est un algorithme garantissant l'exclusion mutuelle, et l'absence de famine et d'interbloquage dans une section critique basé sur de l'attente active.

Pseudo-code

```
tour <- 1-id
veut_renter[id] <- vrai
tant que tour != id et veut_rentrer[1-id]
  attendre
veut_rentrer[id] <- faux
```

8.2 Algorithme de la boulangerie de Lamport

Principe

L'algorithme de la **boulangerie de Lamport** est une méthode pour garantir l'exclusion mutuelle, et l'absence de famine et d'interbloquage d'une section critique, basée sur une attente active.

Son fonctionnement est très imagé, imaginons une boulangerie avec une file d'attente. Un individu, voulant acheter son pain, se mettera dans la file d'attente, mais avant cela, il se voit attribuer un ticket, le démarquant des autres, on prend un ticket plus grand que les autres d'où l'algorithme ci-dessous.

Pseudo-code

```
ticket <- tableau d'entiers (initialisé à -1)
calcul <- tableau de booléens (initialisé à faux

entrée: l'identifiant id du fil

max <- 0
pour i allant de 1 à n faire
  si ticket[i] > max alors
    max <- ticket[i]
ticket[id] <- max + 1
calcul[id] <- faux
```

Le problème étant que, si deux fils rentre en même temps dans la section critique, ils vont être confrontés à un problème, car ils pourront avoir le même ticket!
On verra que ce problème n'a pas d'incidence, car on considérera alors dans ce cas que le fil avec le plus grand identifiant rentrera en priorité dans la section critique.
On fait face tout de même à un autre problème. Si un fil vient à calculer vite son ticket alors qu'un autre vient à peine de commencer, et n'a donc pas finit son calcul, il pourrait avoir le même ticket, mais aussi celui ayant calculer plus vite que l'autre purrait rentrer (même sans avoir la priorité de l'identifiant) rentrer en section critique! Pour cela, il est impératif de mettre en place une barrière de potentiel, nécessaire pour vérifier si tous les fils ont bien tous calculés leur ticket avant de rentrer dans le test de la section critique.

```
pour i allant de 1 à n faire
  tant que calcul[i] attendre
```

Puis,

```
pour i allant de 1 à n faire
  tant que ticket[i] < ticket[id] ou (ticket[i] = ticket[id] et i
     < id)
    attendre
```

Finalement, le fil jète son ticket à la fin (pour pouvoir permettre aux autres de passer)

```
ticket[id] <- -1
```

9 Logique

9.1 Algorithme de Quine

Principe

L'algorithme de **Quine** à pour but de répondre au problème suivant : étant donnée une formule logique φ, est elle satisfiable ?

Pour cela, on construit un arbre binaire, dit de déduction, où l'on évalue à chaque étage une nouvelle variable à vrai ou faux.

Ainsi, φ est satisfiable si et seulement si une feuille est étiqueté par $\top$.

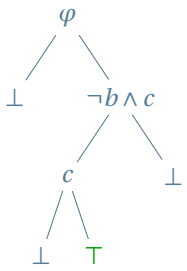
Pseudo-code

```
entrée : formule φ

pour toute variable x de φ restante :
  tester récursivement si φ[x ← ⊤] est satisfiable
  tester récursivement si φ[x ← ⊥] est satisfiable
```

Un exemple

Prenons la formule logique suivante : $\varphi = (a \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee c) \wedge (a \vee c)$.
On construit ainsi l'arbre de déduction suivant (en évaluant la variable par ordre alphabétique et à gauche à faux et à droite à vrai) :



Ainsi la formule φ est satisfiable.

Complexité

La complexité de l'algorithme de **Quine** est celle d'un algorithme de force brute dans le pire des cas, soit en $\mathcal{O}(2^n)$, avec n le nombre de variables composant φ.

10 SQL

10.1 Commandes en vrac

`Select` : ne conserve que certaines colonnes d'une table.

`From` : indique la table d'où l'on extrait.

`As` : renommer les tables.

`Where` : filtrer les entrées répondant à certains critères.

`Distinct` : évite les répétitions.

`Group By` : rassemble plusieurs lignes si elles ont la même valeur dans une certaine colonne.

`Order By` : trier.

`Between` : permet de faire un test d'encadrement.

`Join` : permet de faire correspondre les données de plusieurs tables.

`Having` : similaire au where mais pour le `Group By`, permet de ne rassembler que certaines lignes.

`Limit` et `Offset` : permet de restreindre la taille de la réponse.

`Left Join` et `Right Join` : réalise le produit cartésien puis conserve les lignes où il y a correspondance.

`Insert Into` : modifie.

`In` : teste l'appartenance à une table.

`Union`, `Intersect` : réalise des opérations ensemblistes sur des tables.

`Except` : privé de.

`Like` : teste si une colonne correspond à un modèle.

`Avg` : moyenne.

`Asc` : permet de trier par ordre croissant.

`Desc` : permet de trier par ordre décroissant.

`Upper` : majuscule.

Une **clef primaire** est propre à une table et unique à chaque ligne.

Une **cleft secondaire** est une clef primaire d'une autre table.

11 Structures

11.1 Tables de hachage

Principe

Une **table de hachage** est une structure de données qui permet une association clé-valeur permettant de retrouver une clé donnée très rapidement en la cherchant à un emplacement de la table correspondant au résultat d'une fonction de hachage calculée en $\mathcal{O}(1)$.

Soit U l'ensemble des valeurs possibles et K celui des valeurs effectivement utilisées, on défini $h : U \mapsto \{0, ..., |K|\}$ la **fonction de hachage** telle que l'élément i se trouve à la case $h(i)$ de la **table de hachage**.

Bonne nouvelle : on gère $|K|$ au lieu de $|U|$.
Mauvaise nouvelle : plusieurs valeurs peuvent avoir la même valeur de hachage : on parle de collision.

Régler les problèmes de collision

Le chaînage

Avec cette technique, on place dans une liste chaînée tous les éléments ayant la même valeur de hachage : pour rechercher un élément, il suffit à présent de parcourir la liste chainée dont l'accès se trouve dans la case d'indice $h(x)$. L'avantage est l'aisance de l'implémentation, et de plus, il n'y a pas de limitation (théorique) au nombre de clés. Toutefois, il faut veiller à prendre un tableau assez grand pour que la taille des listes reste réduite, le but étant de minimiser les parcours.

L'adressage ouvert

L'idée est la suivante : lorsqu'on veut utiliser un emplacement de la table et que celui-ci est occupé, on va voir ailleurs. Dans la technique la plus simple, on va simplement continuer le parcours du tableau, à partir de la position de départ, à la recherche d'un « trou ». Pour la recherche, on procède de même : la fonction de hachage nous indique où commencer à chercher et on poursuit jusqu'à trouver l'élément ou un « trou ».

Complexité

le chaînage :

`insertion` s'execute en $\Theta(1)$
`recherche` s'execute en $\Theta(1+\alpha)$
`suppression` s'execute en $\Theta(1+\alpha)$

l'adressage ouvert :

`recherche` s'execute en $\mathcal{O}\left(\frac{1}{1+\alpha}\right)$

Avec $\alpha := \frac{n}{m}$ le facteur de remplissage de la **table de hachage** T de m cases contenant n éléments.

11.2 Arbres binaires de recherches (ABR)

Principe

Un **arbre binaire de recherche** (ou **ABR**) est une structure de données prenant la forme d'un arbre binaire dont tous les nœuds sont d'étiquette supérieur à leur fils gauche et inférieure au fils droit.

Pseudo-code

On donne les algorithmes des trois opérations principales dans le cas d'un tas-min.

rechercher :

```
entrée : un ABR A, une etiquette e

recherche(A, e)
  si A = ∅
    renvoyer faux
  sinon
    A = (x, fils_gauche, fils_droit)
    si x = e
      renvoyer vrai
    sinon si e<x
      renvoyer recherche(fils_gauche, e)
    sinon
      renvoyer recherche(fils_droit, e)
```

insertion :

```
entrée : un ABR A, une etiquette e

insertion(A, e)
  si A = ∅
    retourner (e, _, _)
  sinon
    A = (x,fils_gauche, fils_droit)
    si e < x
      retourner (x,insertion(fils_gauche,e),fils_droit)
    sinon
      retourner (x,fils_gauche,insertion(fils_droit,e))
```

suppression :

```
entrée : un ABR A, une étiquette e

suppression(A, e)
  si A = ∅
    retourner ∅
  sinon A = (x,fils_gauche,fils_droit)
    si e > x
      retourner (x,fils_gauche,suppression(fils_droit,e))
    si e < x
      retourner (x,suppression(fils_gauche,e),fils_droit)
    sinon
      si fils_gauche = ∅ et fils_droit = ∅
        retourner ∅
      si fils_gauche = ∅
        retourner fils_droit
      si fils_droit = ∅
        retourner fils_gauche
      sinon
        y = max(fils_gauche)
        retourner (y,suppression(fils_gauche,y),fils_droit)
```

Complexité

En considérant des **ABR** équilibrés,

`recherche` s'execute en $\mathcal{O}(\log(n))$,
`insertion` s'execute en $\mathcal{O}(n)$ dans le pire des cas et $\mathcal{O}(\log(n))$ dans le cas moyen,
`suppression` s'execute en $\mathcal{O}(n)$ dans le pire des cas.

11.3 Tas binaire

Principe

Un **tas binaire** est une structure de données servant par exemple à l'implémentation de files de priorités que l'on représente par un arbre binaire vérifiant deux contraintes :

- c'est un arbre binaire complet : tous les niveaux sont entièrement remplis sauf éventuellement le dernier qui l'est de gauche à droite,
- c'est un tas : l'étiquette d'un nœud est inférieur (respectivement supérieure) à celle des ses fils pour un tas-min (respectivement un tas-max).

On l'implémente à l'aide d'un tableau comme suit :

- la racine est à la case 0,
- étant donné un nœud à l'indice i, sont fils gauche est en $2i+1$ et le droit en $2i+2$
- étant donné un nœud à l'indice $i>0$, sont père est en $\lfloor \frac{i-1}{2} \rfloor$

Algorithme

On donne les algorithmes des trois opérations principales dans le cas d'un tas-min.

ajouter :

considérons que l'on souhaite ajouter x à notre tas

- on insère x à la prochaine position libre (en bas à gauche du tas),
- tant que x est d'étiquette inférieur à celle de son père, l'échanger avec celui-ci (percoler vers le haut).

retirer :

on souhaite retirer la racine de notre tas

- on supprime la racine que l'on remplace par le nœud x en dernière place de l'arbre (en bas à gauche du tas),
- tant que x est d'étiquette supérieure à l'un de ses fils, l'échanger avec celui de plus petite étiquette (percoler vers le bas).

Complexité

`ajouter` s'execute en $\mathcal{O}(\log(n))$,
`retirer` s'execute en $\mathcal{O}(\log(n))$,
`construire` s'execute en $\mathcal{O}(\log(n))$.

11.4 Unir et trouver

Principe

Unir et trouver (ou **union-find** en anglais) est un structure de données qui répond à deux opérations : trouver à quelle classe d'équivalence appartient un élément donné, et unir deux sous-ensembles.

Cela est utile par exemple pour minimiser la complexité de la représentation de l'arbre dans l'algorithme de Kruskal.

Exemples d'implémentation

Listes chaînées

On utilise des listes chaînés dont le premier élément est représentant, ce qui permet d'executer `unir` en $\mathcal{O}(1)$ mais `rechercher` en $\Omega(n)$, pour n le nombre d'éléments.

Forêts

Ici, chaque classe est un arbre dont la racine est le représentant. On unie deux classes en attachant le plus petit arbre à la racine de l'autre.

Les opérations `créer`, `rechercher` et `unir` ont donc une complexité amortie en $\mathcal{O}(\log(n))$ dans le pire des cas, et en $\mathcal{O}(1)$ dans le meilleur cas.

Voir la section complexité amortie

12 Rappels de cours

12.1 Mots et langages

Σ est appelé un **alphabet**.
$a \in \Sigma$ est une **lettre**.
$(u_i)_{i \in I} \in \Sigma^I$ est un **mot**.
On appelle **langage** sur Σ un ensemble de mots sur l'alphabet Σ.

Expressions régulières (ou rationnelles)

Les expressions régulières sur Σ sont définies (syntaxiquement, i.e. sans sens) inductivement :

- $a \in \Sigma$ est une expression régulière
- ε est une expression régulière
- $\emptyset$ est une expression régulière
- $e_1 \mid e_2$ est une expression régulière si e_1 et e_2 en sont
- $e_1 e_2$ est une expression régulière si e_1 et e_2 en sont
- e^* est une expression régulière si e en est une

On dit d'un langage qu'il est **régulier** s'il existe une expression qui le caractérise.
Il existe des langages non réguliers.
On définit **Reg**(Σ) comme l'ensemble des langages réguliers sur Σ.

Propositions

> Si L est un langage de taille finie, il existe une expression régulière e telle que $L(e) = L$.
>
> Reg(Σ) est le plus petit ensemble de langages contenant $\{\varepsilon\}$, $\emptyset$ et $\{a\}$ qui soit stable par union, concaténation et passage à l'étoile de Kleene.

12.2 Automates

On appelle **automate fini déterministe** un quintuplet $(\Sigma, Q, q_0, F, \delta)$ où

- Σ est l'alphabet,
- Q est l'ensemble des états,
- q_0 est l'état initial,
- $F \subset Q$ est l'ensemble des états finaux,
- $\delta : Q \times \Sigma \mapsto Q$ est la fonction de transition (pouvant être définie partiellement), que l'on étend à $\delta^* : Q \times \Sigma^* \mapsto Q$, définie par :
  - $\forall q \in Q \quad \delta^*(q, \varepsilon) = q$
  - $\forall q \in Q \quad \forall a \in \Sigma \quad \forall m \in \Sigma^* \quad \delta^*(q, am) = \delta^*(\delta(q, a), m)$.

Un mot m est **accepté** par l'automate $A = (\Sigma, Q, q_0, F, \delta)$ si $\delta^*(q_0, m) \in F$.
Un état q est **accessible** s'il existe un mot m tel que $\delta^*(q_0, m) = q$.
Un état q est **co-accessible** s'il existe un mot m tel que $\delta^*(q, m) \in F$.
Un état est dit **utile** s'il est accessible et co-accessible.
Un automate est dit **émmondé** si tous ses états sont utiles.
Un automate est **complet** si sa fonction de transition est définie sur tout $Q \times \Sigma$.
Pour completer un automate, on peut ajouter un état q_p appelé **puit** qui devient l'état d'arrivé de toutes les transitions non définies.
Un automate est **généralisé** si ses transitions sont étiquetées par des expressions régulières.
Le **langage des mots acceptés** est celui des mots m tels que $\delta^*(q_0, m)$ est définie est appartient à F.
Un langage est **locale** si

$$L \setminus \{\varepsilon\} = \left(P(L)\Sigma^* \cap \Sigma^* D(L)\right) \setminus \left(\Sigma^*(\Sigma^2 \setminus F(L))\Sigma^*\right).$$

Voir les ensembles $P(L)$, $D(L)$ et $F(L)$

Un **automate non déterministe** est un quintuplet $(\Sigma, Q, I, F, \delta)$ où I est l'ensemble des états initiaux et $\delta : Q \times \Sigma \mapsto \mathscr{P}(Q)$ est la fonction de transition définie par :

- $\forall q \in Q \quad \delta^*(q, \varepsilon) = \{q\}$
- $\forall q \in Q \quad \forall a \in \Sigma \quad \forall m \in \Sigma^* \quad \delta^*(q, am) = \bigcup_{q' \in \delta(q,a)} \delta^*(q', m)$.

Propositions

L'ensemble des langages reconnus par des automates est stable par intersection et passage au miroir.

Si e est une expression régulière linéaire, $L(e)$ est locale.

Les langages réguliers sont stables par passage au complémentaire.

Théorème de Kleene

Les langages reconnus par des automates sont exactement les langes réguliers.

Lemme de l'étoile

Soit L un langage régulier.
Alors il existe $N \in \mathbb{N}$ tel que

$$\forall m \in L \quad |m| > N \implies \exists (x,y,z) \in (\Sigma^*)^3 \mid \begin{cases} m = xyz \\ |xy| \leq N \\ |y| > 0 \\ \forall k \in N \;\; xy^k z \in L \end{cases} .$$

12.3 Grammaires non contextuelles

Une **grammaire** G est un quadruplet (V, S, Σ, R) où

- V est l'ensemble des **symboles non terminaux** (ou **variables**)
- S est le **symbole initial** : $S \in V$
- Σ est l'ensemble des **symboles terminaux**
- R est l'ensemble des **règles de production**, il est composé d'éléments de $V \times (\Sigma \cup V)^*$

On note une règle $(X, m) : X \to m$ et on parle de **dérivation** selon une grammaire G.
On parle de **dérivation immédiate** entre deux mots m_1 et m_2 que l'on note $m_1 \Rightarrow m_2$ lorsque $\exists \alpha, \beta, \gamma, X \mid m_1 = \alpha X \beta \ \wedge \ m_2 = \alpha \gamma \beta \ \wedge \ (X, \gamma) \in R$.
On note $\Rightarrow^*$ la dérivation entre deux mots, que l'on appelle **fermeture réflexive et transitive** de $\Rightarrow$ définie ainsi :

$$\forall m \in \Sigma^* \quad m \Rightarrow^* m$$

$$\forall (m_1, m_2, m_3) \in (\Sigma^*)^3 \quad \text{si} \begin{cases} m_1 \Rightarrow^* m_2 \\ m_2 \Rightarrow m_3 \end{cases} \quad \text{alors} \quad m_1 \Rightarrow^* m_3.$$

Si $G = (V, S, \Sigma, R)$ est une grammaire, on note $L(G) := \{m \in \Sigma^* \mid S \Rightarrow^* m\}$ le **langage engendré** par la grammaire.

Arbres d'analyse syntaxique

Si un mot possède deux arbres d'analyse syntaxique différents, on dit de la grammaire qu'elle est ambigue.

Forme normale de Chomsky (FNC)

Une grammaire G est sous **FNC** si toutes ses règles sont d'une des formes suivantes :

- $X \to YZ$ avec S le symbole initial
- $X \to x$ X, Y et Z des non terminaux (non nécessairement différents)
- $X \to \varepsilon$ $x \to$ un terminal

  si cette règle est présente, il est demandé que l'axiome n'apparaisse jamais dans le membre droit d'une règle.

Propositions

Les langages réguliers sont inclus dans les langages engendrés par des grammaires non contextuelles.

Il existe des langages engendrés par des grammaires non contextuelles tels que la grammaire est forcement ambigu.

Si G est une grammaire non contextuelle, il existe G' sous FNC telle que $L(G) = L(G')$.

12.4 Classes de complexités

Les classes de complexité sont des ensembles de problèmes que l'on rassemble en fonction de leur difficultés selon plusieurs critères.

La **classe P** correspond aux problèmes de décision qui peuvent être résolus en temps polynomial en la taille de leur entré.
La **classe NP** correspond aux problèmes pour lesquels on peut vérifier une solution en temps polynomial.
Une instance I d'un problème P est dite **positive** si la réponse à la question est oui.
Un algorithme V est un **vérificateur** de P si :
I instance positive ssi il existe c tel que $V(I,c)$ renvoie vrai.
Un problème P **appartient à NP** si :

- il existe un vérificateur V qui s'execute en temps polynomial en la taille de ses entrées,
- il existe un polynôme R tel que la taille d'un certificat valide est bornée par $R(|I|)$.

On appelle **réduction** de P_1 vers P_2 une fonction f qui étant donnée une instance de P_1 renvoie une instance de P_2 et vérifie la propriété suivante :

$$\text{I instance positive de } P_1 \Rightarrow f(I) \text{ instance positive de } P_2.$$

On dit d'une réduction qu'elle est **polynomiale** si elle se calcul en temps polynomial en la taille de son entrée.
Un problème P est dit **NP-difficile** si pour tout $P' \in$ NP, P' se réduit polynomialement à P.
Un problème P est dit **NP-complet** s'il est NP-difficile et qu'il appartient à NP.

Propositions

P $\subset$ NP

Si P_2 est dans P et P_1 se réduit polynomialement à P_2, P_1 est dans P.

Si P_1 se réduit polynomialement à P_2, alors $P_2 \in$ NP $\Rightarrow$ $P_1 \in$ NP.

Si P_1 est NP-complet et se réduit polynomialement à P_2, alors P_2 est NP-difficile.

Théorème de Cook

Le problème SAT est NP-difficile.

Corollaires

SAT-FNC est NP difficile.
SAT et SAT-FNC est NP-complets.

12.5 Décidabilité

Etant donné un problème de décision P, on dit que P est **décidable** s'il existe un algorithme qui s'arrête sur toutes les instances et renvoie vrai ssi elle est positive.
S'il n'est pas décidable, le problème est dit **indécidable**.

Propositions

Il existe des problèmes indécidables.

S'il existe une réduction de P_1 à P_2 et P_2 est décidable, alors P_1 l'est aussi.

Théorèmes

Il existe un algorithme qui étant donné le code d'une fonction f et un paramètre x, simule $f(x)$.

`Arrêt :`
`instance : le code d'une fonction` f `, un paramètre` x
`question : la fonction` f `s'arrête-t'elle sur l'entrée` x `?`

Arrêt est indécidable.

Preuve

On suppose qu'il existe une fonction `arrêt` de type `string` → `string` → `bool` qui décide de ce problème.

On définit `paradoxe` : `string` → `bool` de la façon suivante :

```
let rec paradoxe code_f =
  if arret code_f code_f
  then paradoxe code_f
  else true
```

Dans le cas où f s'arrête sur son code, paradoxe ne s'arrête pas sur $\texttt{code}_f$.
Est-ce que `paradoxe` s'arrête son propre code, noté c.

`paradoxe` s'arrête sur c $\iff$ arret c c renvoie vrai
$\iff$ `paradoxe` c fait un appel récursif à `paradoxe` c
$\iff$ `paradoxe` ne s'arrête pas sur c

Il n'existe pas de telle fonction `arrêt`.

□

12.6 Couplages

Définitions

Un **couplage** dans un graphe biparti est un ensemble d'arêtes C tel que

$$\forall(\{a_1,b_1\},\{a_2,b_2\}) \in C^2 \quad \{a_1,b_1\} \cap \{a_2,b_2\} = \emptyset.$$

Un couplage est **maximal** s'il n'est inclu dans aucun autre couplage.
Un couplage est **maximum** s'il est de cardinal maximum.
Un couplage est **parfait** si chaque sommet du graphe est extrémité d'une arête du couplage.

Un chemin est **alternant** s'il alterne entre arêtes de C et de $A\backslash C$.
Un chemin est **améliorant** s'il est alternant que ses deux extrémités ne sont pas extrémités d'une arête de C.

Théorème de Berge

> Un couplage est maximum si et seulement si il n'admet pas de chemin améliorant.

Pseudo-code

Un premier algorithme permet d'obtenir un couplage maximum (en retirant les chemins améliorants donc)

```
entré : graphe biparti G=(S,A)

C <- ∅
tant qu'il existe C_h chemin améliorant
  C <- C\C_h ∪ C_h\C
renvoyer C
```

Pour trouver un chemin améliorant, on part d'un sommet qui n'est pas extrémité d'une arête de C et on réalise un parcours en profondeur où l'on impose d'alterner entre arêtes de C et de $A\backslash C$. On s'arrête si on trouve un sommet qui n'est pas extrémité d'une arête de C.

```
entré : graphe biparti G=(S,A), liste des sommets deja vus
        deja_vus, sommet s, booléen b, couplage C

parcours(G, deja_vus, s, b, C)
  deja_vus[s] <- vrai
  si s n'est pas extrémité d'un arête de C
    renvoyer vrai
  pour chaque voisin t de s
    si (b et (s,t)∉C) ou (¬b et (s,t)∈C)
      si parcours(G, deja_vus, t, ¬b, C)
        renvoyer vrai
  renvoyer faux
```

Complexité

La complexité de l'algorithme de recherche d'un couplage maximal est en $\mathcal{O}(|S|\cdot(|A|+|S|))$.

12.7 λ-approximation

Définition

Si on étudie un problème d'optimisation P où l'on cherche à minimiser une fonction f, on dit d'un algorithme A qu'il s'agit d'une **λ-approximation** si pour toute instance I, on a

$$\boxed{\lambda f(s_{\text{opt}}) \geq f(s_A)}$$

où s_{opt} est la solution optimale et s_A celle de l'algorithme A.

$f(s_A) \in [f(s_{\text{opt}}); \lambda f(s_{\text{opt}})]$: plus λ est proche de 1, plus l'algorithme est proche de renvoyer une solution optimale.

On parle d'approximation à facteur constant car λ ne dépend pas de l'instance.

Les méthodes s'étendent aux problèmes de maximisation. On parle aussi de **λ-approximation** dans la cas où pour toute instance,

$$\boxed{f(s_{\text{opt}}) \leq \lambda f(s_A)}$$

$$\text{i.e. } f[\frac{1}{\lambda}; f(s_{\text{opt}})]$$

Toutes les valeurs de λ sont considérés supérieurs à 1.

12.8 Déduction naturelle

Rappels

Une **formule logique** est définie inductivement :

- $\bot$ (faux)
- $\top$ (vrai)
- x (où x est un littéral)
- $\varphi_1 \wedge \varphi_2$ (où φ_1 et φ_2 sont des formules logiques)
- $\varphi_1 \vee \varphi_2$ (où φ_1 et φ_2 sont des formules logiques)
- $\neg\varphi$ (où φ est une formule logique)
- $\varphi_1 \Rightarrow \varphi_2$ (où φ_1 et φ_2 sont des formules logiques)

Une **valuation** est une application qui à une formule logique associe une valeur de vérité.
Un **modèle** est une valuation qui satisfait une formule, on note $\mathcal{M}(\varphi)$ l'ensemble des modèles de φ.
Une formule φ est **satisfiable** si elle admet un modèle.
φ **implique sémantiquement** ψ si $\mathcal{M}(\varphi) \subset \mathcal{M}(\psi)$, on note alors $\varphi \models \psi$.
Une **tautologie** est une formule que toutes les valuations satisfont.
Deux formules φ et ψ sont **équivalentes** si pour toute valuation v, on a $v(\varphi) = v(\psi)$ et on note alors $\varphi \equiv \psi$.

Les arbres de preuve

Un **séquent** est un couple (Γ, φ) où Γ est un ensemble de formules logiques et φ est une formule logique, on les notes $\Gamma \vdash \varphi$.

Une **règle d'inférence** est un $n+1$ uplet de séquents $(P_1, ..., P_n, C)$ où $P_1, ..., P_n$ sont les prémisses et C la conclusion, que l'on note $\dfrac{P_1 \quad \cdots \quad P_n}{C}$.

Etant donné un système S de règles d'inférence, on définit un **arbre de preuve** de manière inductive :

- une application d'un axiome à un séquent ;
- $\dfrac{A_1 \quad \cdots \quad A_n}{C}$ où $A_1, ..., A_n$ sont les arbres de preuve respectifs des séquents $P_1, ..., P_n$.

Systèmes déductifs

Un **système déductif** est un ensemble de règles d'inférences.
Ci dessous les différents systèmes déductifs au programme :

logique **minimale**	$\wedge, \vee, \neg, \Rightarrow, \top_i, aff$
logique **intuitionniste**	logique minimale + $\bot_e$
logique **classique**	logique minimale + abs

Une règle est dite **dérivable** dans un système déductif si lorsque l'on rajoute ses prémisses comme axiomes, la conclusion est prouvable. Autrement dit, on peut la construire à l'aide des règles du système déductif.
On parle de **règle gauche** lorsqu'une règle ne modifie que le contexte.
Deux systèmes déductifs sont **équivalents** si tout ce qui est prouvable dans l'un l'est aussi dans l'autre.

Un système déductif est dit :

- **correct** si $\Gamma \vdash \varphi$ prouvable implique $\Gamma \models \varphi$;
- **complet** si $\Gamma \models \varphi$ implique $\Gamma \vdash \varphi$ prouvable ;
- **cohérent** si l'on a pas $\vdash \bot$

Règles classiques

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_i) \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow_e)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge_i) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge_{eg}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge_{ed})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee_{ig}) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee_{id}) \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee_e)$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ (\neg_i) \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\ (\neg_e)$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ (aff)$$

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ (abs)$$

$$\frac{}{\Gamma \vdash \top}\ (\top_i) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot_e)$$

$$\frac{}{\Gamma \vdash A \vee \neg A}\ (t_e) \qquad \frac{\Gamma \vdash \neg B \Rightarrow \neg A}{\Gamma \vdash A \Rightarrow B}\ (c) \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ (\neg\neg_e)$$

$$\frac{}{\Gamma \vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}\ (l_p)$$

Propositions

Toute formule φ est équivalente à $\bot$, $\top$ ou φ' n'utilisant ni $\bot$ ni $\top$.

Considérons un système logique L et une règle R dérivable dans L, alors tout séquent prouvable dans $L \cup \{R\}$ est prouvable dans L.

La logique intuitionniste munie du tiers exclu est équivalente à la logique classique.

La logique classique est équivalente à la logique intuitionniste munie de la contraposé, de la double négation ou bien de la loi de Pierce.

Un système déductif correct est cohérent.

Les logiques minimale, classique et intuitionniste sont correctes.

La logique classique est complète, contrairement aux logiques intuitionniste et minimale qui ne permettent pas de prouver $\Gamma \vdash A \vee \neg A$.

Logique du premier ordre

Un **langage du premier ordre** est la donnée :

- d'un ensemble de symboles de fonctions ;
- d'un ensemble de symboles de relations.

Exemple : la théorie des groupes
fonctions : $^{-1}, \star, e$
relation : $=$

Une **formule logique du premier ordre**, étant donné un langage du premier ordre, est définie inductivement :

- une formule **atomique** : des termes liés par une relation ;
- si φ et ψ en sont, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ et $\varphi \Rightarrow \psi$ en sont aussi ;
- $\forall x\ \varphi$ et $\exists x \mid \varphi$ où x est une variable.

On appelle **occurence** d'une variable x dans φ une feuille de φ étiquetée par x.
On dit d'une occurence de x qu'elle est **liée** si elle descend d'un $\forall x$ ou d'un $\exists x$.
Une variable est dite **libre** si l'une de ses occurence n'est pas liée.
Une formule sans variable libre est appelé **formule close**.
La **substitution** d'une variable x par t dans une formule logique du premier ordre φ consiste à remplacer les occurrences libres de x par t.

$$\frac{\Gamma \vdash \varphi[x := t]}{\Gamma \vdash \exists x \mid \varphi}\ (\exists_i) \qquad \frac{\Gamma \vdash \exists x \mid \varphi \qquad \Gamma, \varphi \vdash \psi}{\Gamma \vdash \psi}\ (\exists_e)$$

où x n'est pas libre dans φ ni dans ψ

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x\ \varphi}\ (\forall_i) \qquad \frac{\Gamma \vdash \forall x\ \varphi}{\Gamma \vdash \varphi[x := t]}\ (\forall_e)$$

où x n'est pas libre dans Γ

12.9 Complexité amortie

Principe

L'**analyse amortie** consiste à relativiser le coût élevé de certaines opérations réalisés sur une structure de donnés. Au lieu d'étudier les opérations de manière indépendante, on les étudie en tant qu'une suite d'opérations dont on aimerait connaître la complexité moyenne. Cela permet notamment de rendre compte que la rapidité de certaines opérations est la conséquence de la lenteur d'autres.

Un exemple

Prenons l'exemple des tableaux dynamiques, c'est-à-dire des tableaux défini par leurs tailles, leurs capacités et le tableau en lui même.

Méthode moyenne (ou des agrégats)

Le principe est de sommer le coût de n opérations puis diviser par n pour obtenir le coût amorti.

Ici, on insère n éléments dans le tableau, le coût est de

$$\begin{cases} \mathcal{O}(1) & \text{s'il reste de la place} \\ \mathcal{O}(|T|) & \text{sinon} \end{cases}$$

la capacité double à chaque allocation, la prochaine allocation ayant lieu après $|T|$ insertions.

Notons $C(i_k)$ le coût de la k-ème insertion :

$$\begin{aligned} \sum_{k=1}^{n} C(i_k) &= \sum_{k=1}^{n} \left(1 + \sum_{\substack{1 \leq k \leq n \\ k=2^i}} k \right) \\ &\leq n + \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i \\ &\leq n + 2^{\lfloor \log_2 n \rfloor + 1} \\ &\leq 3n \end{aligned}$$

Ainsi $\frac{1}{n} \sum_{k=1}^{n} C(i_k) = \mathcal{O}(1)$.

Méthode du potentiel

Le principe est de définir une fonction Φ qui étant donné notre structure, quantifie à quel point on a réalisé beaucoup d'opérations peu coûteuses ou à quel point on est près d'en réaliser une coûteuse.

Ici, Φ = capacité – taille :

- vaut 0 lorsqu'on doit doubler la capacité du tableau,
- est maximum lorsqu'on vient de doubler,
- diminue à chaque insertion.

Pour chaque opération, on calcule sa complexité amortie en mesurant la variation du potentiel qui a eu lieu

$$\hat{C}(i_k) = C(i_k) - \Delta\Phi.$$

Ici il y a deux cas :

- insertion simple : $\hat{C}(i_k) = 1 - (-1) = 2 = \mathcal{O}(1)$

  car $\Delta\Phi = \text{capacité}_{\text{finale}} - \text{taille}_{\text{finale}} - (\text{capacité}_{\text{initiale}} - \text{taille}_{\text{initiale}}) = \text{taille}_{\text{initiale}} - \text{taille}_{\text{finale}} = -1$
- insertion avec recopiage : $\hat{C}(i_k) = \text{taille-(taille-1)} = \mathcal{O}(1)$

  car $\Delta\Phi = \text{capacité}_{\text{finale}} - \text{taille}_{\text{finale}} - \underbrace{(\text{capacité}_{\text{initiale}} - \text{taille}_{\text{initiale}})}_{=0 \text{ car le tableau est rempli}} = 2\text{taille}_{\text{initiale}} - (\text{taille}_{\text{initiale}} + 1) = \text{taille}_{\text{initiale}} - 1$

Méthode des acomptes (ou du banquier)

Le principe est de donner des sous aux éléments qui pourront les dépenser pour de futurs opérations. Ici à chaque insertion, on donne k pièces à l'élément inséré, ces k pièces doivent financer les opérations futurs du tableau dynamique.

Une simple insertion coûte une pièce.

Doubler la taille du tableau coûte la taille du tableau en pièces (une par recopiage).

Ainsi, avec 4 pièces par insertion, la seconde partie du tableau $T := T_{\frac{1}{2}}\, T_{\frac{2}{2}}$ paye une pièce pour son insertion, puis conserve une pièce par élément $T_{\frac{2}{2}}[i+k]$ pour la recopie de l'élément $T_{\frac{1}{2}}[i]$ et enfin une pièce pour son propre recopiage : toutes les opérations sont financées.

Le problème de la méthode des acomptes réside dans la difficulté de choisir un bon coût fictif.

La **complexité amortie** du tableau dynamique est donc en $\mathcal{O}(1)$.

13 Preuves

Preuve (complexité moyenne du tri rapide) Retourner au tri rapide

On note $C_m(N)$ le nombre moyen de comparaisons entre deux éléments d'un tableau de N éléments, par exemple $C_m(0) = 0$, $C_m(1) = 0$, et $C_m(2) = 1$.

On fixe $N \geq 2$:
on réalise déjà $N-1$ comparaisons dans partition. De cela on en déduit le pivot, qui détermine alors les tailles des tableaux après division, or le choix du pivot est uniforme dans le tableau.
Alors

$$C_m(N) = N - 1 + \frac{1}{N}\sum_{k=0}^{N-1}(C_m(k) + C_m(n-k-1)).$$

Puis

$$C_m(N) = N - 1 + \frac{2}{N}\sum_{k=0}^{N-1} C_m(k).$$

De cela, on en déduit le système suivant

$$\begin{cases} NC_m(N) = N(N-1) + 2\sum_{k=0}^{N-1} C_m(k) \\ (N-1)C_m(N-1) = (N-1)(N-2) + 2\sum_{k=0}^{N-2} C_m(k) \end{cases}$$

Puis $NC_m(N) - (N+1)C_m(N-1) = 2(N-1)$

Donc $\frac{C_m(N)}{N+1} - \frac{C_m(N-1)}{N} = \frac{2(N-1)}{N(N+1)}$

Puis par télescopage

$$\frac{C_m(N)}{N+1} - \frac{C_m(2)}{3} = \sum_{k=0}^{N-1}\frac{2(k-1)}{k(k+1)}.$$

Puis

$$\frac{C_m(N)}{N+1} = \frac{1}{3} + 2\sum_{k=0}^{N-1}\frac{1}{k+1} - 2\sum_{k=0}^{N-1}\frac{1}{k(k+1)}.$$

Finalement, en reconnaissant la série harmonique,

$$C_m(N) = \mathcal{O}(NlogN).$$

□