Informatique 1 — sujets de programmation (1999, groupe 3)

Liste des projets

1 Réseaux de transport**

1.1 Objectif

Certains problèmes de transport (de marchandises, de fluides, d'énergie, etc) dans un réseau (routier, de distribution, etc) sont représentables de façon naturelle par des graphes dont les sommets sont des points de passage (sans stockage) et les arcs des trajets entre ces points. On demande d'implémenter un algorithme qui maximise l'utilisation de tels réseaux, c'est-à-dire qui calcule la quantité maximale transportable compte tenu des contraintes.

1.2 Méthode

On modélise un réseau de transport à l'aide d'un graphe $\mathcal{G} = (S, A)$, muni d'une fonction de capacité $c : S \times S \to \mathbf{N}$, telle que $c(x, y) = 0$ si $(x, y) \notin A$. On suppose qu'un sommet *source* s et un sommet *puit* t ont été désignés et que tout sommet du graphe se trouve sur un chemin de s à t.

On appelle *flot* une fonction $f : S \times S \to \mathbf{Z}$ telle que (contraintes de capacité et de conservation, Cf. les lois de Kirchhoff) :

1. quels que soient $x,\, y \in S,\ f(x,y) \leq c(c,y)$
2. quels que soient $x,\, y \in S,\ f(x,y) = -f(y,x)$
3. quel que soit $x \neq s,t,\ \sum_{y \in S} f(x,y) = 0$

La *valeur* du flot f est le nombre $|f| = \sum_{y \in S} f(s,y)$; elle mesure la quantité maximale transportable de s vers t, compte tenu des contraintes.

Le problème est de déterminer un flot de valeur maximale. Sa résolution fait appel à la notion de réseau résiduel d'un flot. Étant donné un réseau de graphe $\mathcal{G}$) et de capacité c et un flot f, on définit $c_f(x,y) = c(x,y) - f(x,y)$, pour $(x,y) \in S \times S$; le réseau résiduel de f est défini par le graphe $(\mathcal{G}_f)$ ayant les mêmes sommets que $\mathcal{G}$ et dont l'ensemble des arcs est $A_f = \{(x,y) \in S \times S; c_f(x,y) > 0\}$, et par la capacité c_f.

Un chemin d'*augmentation de f* dans un réseau est un chemin simple (c'est-à-dire sans boucle) de s vers t dans le graphe résiduel $\mathcal{G}_f$. La capacité d'un tel chemin p est la quantité maximale transportable le long de ce chemin : $c(p) = \min\{c_f(x,y); (x,y) \in p\}$. Un tel chemin permet d'augmenter f en définissant

$$
\begin{aligned}
f'(x,y) &= f(x,y) + c(p) \quad \text{si } (x,y) \in p \\
f'(x,y) &= f(x,y) - c(p) \quad \text{si } (y,x) \in p \\
f'(x,y) &= f(x,y) \quad \text{sinon}
\end{aligned}
$$

Il s'agit bien d'une augmentation de f au sens où l'on peut vérifier que $|f'| > |f|$. On montre alors qu'un flot est maximum si et seulement si son graphe résiduel ne contient aucun chemin d'augmentation. L'algorithme de Ford-Fulkerson (1956) consiste à augmenter le flot selon un chemin d'augmentation s'il en existe :

- le flot f est initialisé à 0
- (itération) tant qu'il existe un chemin d'augmentation dans $\mathcal{G}_f$, en choisir un de longueur minimum, p, et augmenter f selon p
- s'il n'existe pas de chemin d'augmentation dans $\mathcal{G}_f$, alors f est un flot de valeur maximale

1.3 Résultats demandés

Il faudra d'abord choisir une représentation textuelle des réseaux en vue de sauvegarder leur définition dans un fichier. Il faudra ensuite choisir une structure de données adéquate pour les représenter ; on notera que les graphes résiduels partagent le même ensemble de sommets, mais que l'ensemble des arcs est variable. On pourra alors implémenter l'algorithme de Ford-Fulkerson, le résultat étant la définition d'un flot maximum.

On réalisera également une interface graphique permettant de saisir le réseau et d'afficher le résultat.

En application, on devra pouvoir déterminer un appariement maximum dans un graphe biparti. Le programme résolvant ce problème devra faire appel au programme implémentant l'algorithme de Ford-Fulkerson auquel un réseau convenable sera soumis.

2 Arithmétique en précision arbitraire**

2.1 Objectif

Produire une bibliothèque d'arithmétique sur les grands entiers. On utilisera un système de numération en base $\mathbf{B}$, dont les « chiffres » sont les entiers compris entre 0 et $\mathbf{B}-1$. La notation $[\mathsf{u}_m \dots \mathsf{u}_0]_{\mathbf{B}}$ désigne l'entier $\mathsf{u}_m\mathbf{B}^m + \dots + \mathsf{u}_0$.

Cette bibliothèque doit pouvoir servir à écrire une fonction factorielle sur des entiers arbitrairement grands, comme par exemple :

```
#include "arith.h"

Int factorielle(int n)
{
  Int *p = mk_Int(1);
  int i;
  for (i=1; i<=n; i++) {
    int_Int_mult(i, p, p);
  }
  return *p;
}
```

2.2 Méthode

Il s'agit de programmer des algorithmes que l'on applique d'habitude « à la main ».

La seule opération délicate à programmer est la division euclidienne***. Soit $u = [\mathsf{u}_m \dots \mathsf{u}_0]_{\mathbf{B}}$ et $v = [\mathsf{v}_n \dots \mathsf{v}_0]_{\mathbf{B}}$ des grands entiers positifs ; examinons le cas où $m = n+1$, et où $u/v < \mathbf{B}$, c'est-à-dire où le quotient est un chiffre q : on cherche q et r tels que $u = \mathsf{q}v + r$, avec $r < v$.

On calcule une approximation $\hat{\mathsf{q}}$ de q en utilisant les deux chiffres de gauche de u : $\hat{\mathsf{q}} = (\mathsf{u}_{n+1}\mathbf{B} + \mathsf{u}_n)/\mathsf{v}_n$, si $\mathsf{u}_{n+1} \neq \mathsf{v}_n$, et $\hat{\mathsf{q}} = \mathbf{B} - 1$ sinon. On a toujours $\hat{\mathsf{q}} \geq \mathsf{q}$, et dans le cas où $\mathsf{v}_n \geq \mathbf{B}/2$, on a aussi $\hat{\mathsf{q}} \leq \mathsf{q} + 2$. On dit que le diviseur v est normalisé si $\mathsf{v}_n \geq \mathbf{B}/2$; dans le cas d'un diviseur normalisé, le quotient q vaut $\hat{\mathsf{q}}$, $\hat{\mathsf{q}} - 1$ ou $\hat{\mathsf{q}} - 2$: il suffit de trouver le premier « reste » positif parmi $u - \hat{\mathsf{q}}v \geq 0$, $u - (\hat{\mathsf{q}} - 1)v \geq 0$ et $u - (\hat{\mathsf{q}} - 2)v \geq 0$ pour déterminer exactement le quotient q et le reste r.

Il est donc nécessaire de commencer par une normalisation ; celle-ci consiste à calculer $\alpha = \mathbf{B}/(\mathsf{v}_n + 1)$ et à remplacer $u = [\mathsf{u}_m \dots \mathsf{u}_0]_\mathbf{B}$ par $u' = \alpha u = [\mathsf{u}'_{m+1} \dots \mathsf{u}'_0]_\mathbf{B}$ (notez le nouveau chiffre u'_{m+1}, qui est éventuellement nul), et $v = [\mathsf{v}_n \dots \mathsf{v}_0]_\mathbf{B}$ par $v' = \alpha v = [\mathsf{v}'_n \dots \mathsf{v}'_0]_\mathbf{B}$. Une fois le quotient q' et le reste r' de la divioisn de u' par v' obtenus, il suffira de poser $q' = q$ et $r = r'/\alpha$.

Une fois les dividende et diviseurs normalisés, les chiffres $\mathsf{q}_{m-n}, \dots, \mathsf{q}_0$ du quotient sont succesivement déterminés à l'aide de la méthode esquissée ci-dessus, de façon itérative, en remplaçant après chaque chiffre calculé q_i, le dividende u par $u - \mathsf{q}\mathbf{B}^i v$.

2.3 Résultats demandés

Plusieurs implémentations sont possibles. On devra notamment choisir la base $\mathbf{B}$ de numération, compte tenu de différents critères à expliciter et la structure de données (tableau, bloc dynamique, liste chaînée).

On commencera par définir un type `Nat` pour les entiers naturels, et un type `Int` pour les entiers rationnels (appelés signés par les informaticiens). Le signe et la taille d'un entier rationnel devront être facilement accesibles. La bibliothèque devra contenir les fonctions suivantes :

- les opérations arithmétiques (addition, soustraction, multiplication, multiplication par une puissance de la base, division euclidienne et modulo) et booléennes (égalité, inégalités)
- des conversions, quand c'est possible, entre `int`, `Nat` et `Int`
- éventuellement, une opération de clonage.
- des entrées sorties (construction ou lecture d'une chaîne de caractères formée de chiffres décimaux, et éventuellement d'espaces pour la lisibilité)

Les fonctions seront rassemblées dans une bibliothèque `libarith.a`, créée à l'aide de la commande `ar` d'Unix. On proposera des utilisations de cette bibliothèque (par exemple, détermination de grands nombres premiers).

3 Ordonnancement de projet*

3.1 Objectif

Certains problèmes d'ordonnancement de projet sont représentables à l'aide d'un graphe dont les sommets sont des événements et les arcs des tâches, ceux-ci étant étiquetés par leur durée ; le sommet initial et final d'un arc représente, respectivement, l'événement de début et de fin d'exécution de la tâche. On cherche à déterminer les tâches critiques, c'est-à-dire celles dont un retard est répercuté sur l'ensemble du projet.

3.2 Méthode

On modélise un projet par un graphe $\mathcal{G} = (S, A)$ et une fonction de durée $d : A \to \mathbf{N}$. On suppose qu'un sommet initial s et un sommet final t ont été désignés. La longueur d'un chemin est la somme des durées des arcs le composant. Un chemin *critique* est un chemin de s à t de longueur maximale. S'n'existe pas de chemin critique, le projet n'est pas réalisable. Le temps minimal requis pour l'exécution d'un projet réalisable est égal à la longueur d'un chemin critique ; c'est la durée du projet. Les tâches figurant sur un chemin critique sont aussi appelées critiques, ce qui s'explique par le fait que tout retard d'exécution d'une tâche critique retarde d'autant l'exécution du projet.

Une *exécution* du projet est définie par une fonction de date de début d'exécution $t : A \to \mathbf{N}$. Si le projet est réalisable, on définit l'exécution *au plus tôt*, $t_{\min}$, et l'exécution *au plus tard*, $t_{\max}$, : $t_{\min}(x, y)$ est la longueur maximale des chemins de s à x, $t_{\max}(x, y)$ est la date la plus tardive pour commencer la tâche (x, y) telle que la durée totale de l'exécution soit la durée du projet. On notera qu'une tâche (x, y) est critique si $t_{\min}(x, y) = t_{\max}(x, y)$. Il est plus simple de calculer des fonctions $t_{\min}, t_{\max} : S \to \mathbf{N}$, puis d'obtenir $t_{\min}(x, y) = t_{\min}(x)$ et $t_{\max}(x, y) = t_{\max}(y) - d(x, y)$. Ces dates sont obtenues par :

$$
\begin{aligned}
t_{\min}(y) &= \max_{x \to y}(t_{\min}(x) + d(x, y)) \\
t_{\max}(x) &= \max_{x \to y}(t_{\max}(y) - d(x, y))
\end{aligned}
$$

L'algorithme consiste, dans une première phase, à calculer les dates $t_{\min}$, initialisée à 0, à partir du sommet initial s, en procédant par un tri topologique du graphe. Dans une seconde phase, $t_{\max}$ est calculée à partir du sommet final, en l'initialisant à l'aide de $t_{\min}(t)$ (maintenant connu), par un tri topologique en ordre inverse.

3.3 Résultats demandés

Il faudra d'abord traiter une représentation textuelle des tâches du projet de la forme suivante, qui spécifie les contraintes d'ordre et de durée :

```
caleçon < pantalon
pantalon < ceinture
chaussettes < chaussures
pantalon < chaussures
chemise < ceinture
chemise < cravate
cravate < veste
chaussettes : 2
caleçon : 1
chemise : 6
pantalon : 4
cravate : 20
...
```

Il faudra ensuite choisir une structure de données adéquate pour les représenter.

On pourra alors déterminer si le projet est réalisable et dans le cas positif, effectuer le calcul de sa durée, des dates au plus tôt et au plus tard, et dire quelles sont les tâches critiques.

On réalisera également une interface graphique permettant de saisir le réseau et d'afficher le résultat.

4 Compression de Huffman**

4.1 Objectif

Écrire un programme capable de réduire la taille d'un fichier (compression) par un codage de Huffman et un programme capable de reconstituer le fichier initial à partir du fichier codé (décompression).

4.2 Méthode

On cherche à coder, par une suite de 0 et de 1 aussi courte que possible, un fichier composé d'octets quelconques (pas nécessairement des caractères), à l'aide d'un codage de Huffman. Le langage source est donc l'ensemble des 256 octets, auquel on ajoute un symbole pour marquer la fin d'un fichier comprimé, soit en tout 257 symboles.

Le principe de l'algorithme est décrit dans le cours, § 75. Voici quelques précisions sur les structures de données à implémenter.

L'algorithme de Huffman impose deux lectures du fichier initial : une première pour l'élaboration de l'histogramme des fréquences et une seconde pour le codage proprement dit. La phase de codage utilise l'arbre de Huffman construit à partir de l'histogramme des fréquences.

Les nœuds de l'arbre de Huffman ne contiennent pas le même type d'information selon qu'il s'agit d'un nœud interne ou d'une feuille. Un nœud interne de l'arbre de Huffman n'a besoin de connaître que ses deux fils. Une feuille ne possède par définition pas de fils mais mémorise un des symboles à coder. On utilise cependant un même type de cellules pour tous les nœuds ; elles comprennent, en plus des champs-pointeurs pour les fils, un champ de type `int`, un pour le caractère codé et un pour la fréquence ; le champ-caractère contiendra -1 pour tous les nœuds internes et le caractère codé ou le symbole de fin de fichier pour les feuilles.

L'ensemble des arbres, ou *forêt*, sera implémenté comme une liste chaînée des cellules de racines des arbres, par ordre de fréquences croissantes.

Dans la procédure de construction de l'arbre de Huffman, on ne s'intéresse qu'aux symboles « actifs » ayant une fréquence d'apparition non nulle. À chaque étape de l'algorithme, les actions suivantes sont effectuées :

1. les deux nœuds ayant les fréquences minimum (mais non nulles) sont les deux premières cellules de la liste chaînée
2. fusionner ces deux nœuds en un nœud père dont la fréquence est mise à la somme des fréquences de ses fils,

Pour réaliser la compression d'un fichier, on a besoin de la correspondance entre un symbole et son code de Huffman. Cette correspondance figure dans une table qui associe à chaque symbole son code et la longueur (nombre de bits) de son code. Cette table est construite par une traversée de l'arbre de Huffman.

Pour pouvoir être décodé, le fichier comprimé doit contenir l'arbre de décodage ou bien les données nécessaires à sa reconstruction, c'est-à-dire l'histogramme des fréquences.

La séquence de bits du fichier comprimé est alors décodée à l'aide l'arbre selon le principe décrit en § 26 du cours.

4.3 Résultats demandés

On vous demande de réaliser un programme de compression (resp .décompression) qui prend en argument un fichier à comprimer (décomprimer) selon l'algorithme de Huffman présenté ci-dessus. Le nom du fichier comprimé devra être suffixé par la chaîne *.hfm*.

Vous devrez comprimer des fichiers de différents types (texte français ou anglais, source C, image, exécutable, ...) et de différentes tailles et vous donnerez pour chaque fichier, le taux de compression obtenu qui est le rapport longueur du fichier initial sur longueur du fichier comprimé. Prenez garde de recopier (commande `cp`) le fichier avant de le comprimer car vous risquez de le perdre irrémédiablement si votre programme ne fonctionne pas correctement. Vous tenterez d'en tirer des conclusions.

4.4 Moyens

On devra utiliser deux types de fonctions sur les fichiers :

1. des fonctions « standard », qui opèrent sur un fichier considéré comme un flot d'octets, déclarées dans `iostream`, et,
2. des fonctions qui opèrent sur un fichier considéré comme un flot de bits, déclarées dans `bitio.h`, dont la documentation vous sera donnée ultérieurement

5 Arbre de recouvrement minimum***

5.1 Objectif

Le problème est de connecter par un réseau (de câbles) un ensemble de points ; on connaît les distances entre tous les couples de points entre lesquels on peut installer un câble, et on veut minimiser la longueur totale du câble qui doit être utilisé.

Ce problème a bien d'autres applications. Par exemple, les points peuvent représenter des données (mots, images, plantes, etc) avec une de distance mesurant une notion proximité entre elles (entre deux mots, ou deux images, etc). On veut répartir ces données en deux classes telles que deux données de la même classe soient plus proches que deux données de deux classes différentes, et obtenir en itérant ce partage, une classification (comme en botanique !).

5.2 Méthode

On modélise le problème par un graphe non-orienté $\mathcal{G} = (S, A)$, muni d'une fonction de coût $c : A \to \mathbf{N}$. Une solution du problème est un sous-graphe $\mathcal{G}' = (S, A')$ connexe, sans circuit (autrement dit, c'est un arbre au sens de la théorie des graphes) de coût minimum, ce coût étant $c(\mathcal{G}') = \sum_{a \in A'} c(a)$; une telle solution est appelée un *arbre de recouvrement de coût minimum*. On suppose que le problème a une solution, c'est-à-dire que le graphe $\mathcal{G}$ est connexe.

Il existe plusieurs algorithmes pour résoudre ce problème. Ce sont des algorithmes *gloutons* qui construisent la solution de façon incrémentale en ajoutant une arête à chaque étape.

L'algorithme de Kruskal (1956) utilise une *forêt* $\mathcal{F}$ (c'est-à-dire un ensemble d'arbres disjoints). Initialement, la forêt est formée de tous les sommets du graphe, sans aucune arête. À chaque étape, on ajoute à la forêt une arête choisie de coût minimum, mais seulement si cette arête ne crée pas de circuit. Pour cela, on range l'ensemble des arêtes par ordre de coût croissant. Pour s'assurer que l'on ne crée pas de circuit, c'est-à-dire que l'arête choisie relie deux arbres distincts de $\mathcal{F}$, on organise la forêt sous la forme d'une structure de données dont les trois opérations sont : initialiser, représenter, fusionner. `Initialiser` consiste à créer une forêt consistant en n sommets, sans arête ; `représenter` associe à tout sommet de la forêt un sommet distingué de l'arbre auquel il appartient, sa racine ; on peut alors déterminer si deux sommets appartiennent au même arbre en comparant leurs racines ; `fusionner` deux sommets consiste à remplacer les arbres auxquels ces sommets appartiennent par un arbre unique, en faisant de la racine de l'un un enfant de la racine de l'autre. On implémentera une forêt à l'aide d'une structure chainée à un lien qui pointe vers le parent (contrairement aux arbres binaires où les liens pointent vers les enfants), la racine de chaque arbre pointant vers elle-même. Dans l'opération `fusionner`, on peut choisir de faire de la racine du plus petit des deux arbres un enfant du plus grand. On peut aussi profiter de chaque opération `représenter` pour comprimer l'arbre en faisant du sommet dont on cherche le représentant un enfant de ce représentant.

L'algorithme de Prim (1957) construit incrémentalement un arbre en ajoutant à chaque étape une arête, choisie de coût minimum. Initialement, l'arbre est formé d'un unique sommet r, quelconque. L'algorithme utilise comme structure de données un *tas-min* (Williams, 1964) contenant à chaque étape tous les sommets qui n'appartiennent pas (encore) à l'arbre. Chaque sommet du tas-min est affecté d'un entier `val` qui est le coût minimum d'une arête reliant ce sommet à un sommet de l'arbre courant ; si un sommet n'est pas reliable par une arête à l'arbre courant, cet entier vaut ∞. Chaque sommet est également affecté d'un lien `parent` vers son parent dans l'arbre. Initialement, le tas-min est formé de tous les sommets, `val` étant ∞, sauf pour r, `val` étant 0 et `parent` étant `NULL`. Ensuite, tant que le tas-min est non vide, on en retire un sommet avec `val` minimum ; soit x ce sommet ; pour chaque sommet y adjacent à x, si y est dans le tas-min et si $c(x,y) < y.\mathtt{val}$, alors on fait de x le parent de y et on pose $y.\mathtt{val} = c(x,y)$. Rappelons qu'un tas-min est une structure de données avec les opérations `inserer`, `minimum` et `extraire_min`. L'implémentation usuelle se fait à l'aide d'un arbre étiqueté, chaque nœud de l'arbre ayant une étiquette inférieure ou égale à celles de ses enfants.

5.3 Résultats demandés

Le programme devra pouvoir traiter des problèmes dont les données sont produites aléatoirement (de façon à tester les algorithmes sur des données de grande taille) ainsi que des problèmes saisis de façon interactive, grâce à une interface graphique. Dans le second cas, le résultat, c'est-à-dire l'arbre de recouvrement de coût minimum, doit aussi être affiché graphiquement.

Les deux algorithmes de Kruskal et de Prim sont à implémenter, dans des programmes différents, mais partageant les mêmes fonctions pour les entrées-sorties.

6 Analyse de textes littéraires*

6.1 Objectif

Fournir des outils d'analyse lexicale automatique de textes littéraires.

6.2 Résultats demandés

Construire un programme `analyse` qui prend en argument un nom de fichier `f.txt` et qui produit deux fichiers `f.ind` et `f.fre`. Le premier contient une liste, par ordre alphabétique, des mots contenus dans `f.txt` et de leur fréquence. Le second contient une liste des mots et de leur fréquence, par ordre de fréquence décroissante.

Tester ce programme sur plusieurs textes. Utilisez le résultat de son exécution sur ces textes pour construire une liste de mots non-significatifs (les articles, prépositions usuelles, etc) ; une option `-x` du programme permettra ensuite d'exclure ces mots des listes produites et du calcul des fréquences.

Le programme `analyse` doit aussi accepter une option `-d`, suivie de deux noms de fichiers `f1` et `f2` (sans leur suffixe) ; elle utilise les fichiers `.ind` et `.fre` pour produire une liste `f1_et_f2.ind` des mots présents dans les deux textes et des listes `f1_non_f2.fre f2_non_f1.fre` des mots présents dans un seul des textes. On cherchera ainsi à déterminer le « lexique » propre à une œuvre ou à un auteur.

6.3 Moyens

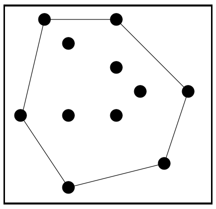
Les textes sont disponibles sur le site Web de l'Association des Bibliophiles Universels : `http://cedric.cnam.fr/ABU/BIB`.

Il faudra utiliser des algorithmes de tri et des structures de données efficaces.

7 Enveloppe convexe d'un ensemble de points*

7.1 Objectif

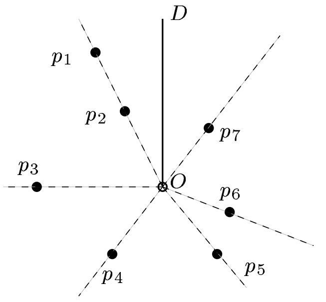
Étant donné un ensemble de points dans le plan, tracer l'enveloppe convexe de cet ensemble est un jeu d'enfant (elle “saute aux yeux”) ! La faire calculer par un ordinateur est un peu plus délicat. Rappelons qu'un ensemble de points S est convexe, si pour tout couple de points (p_1, p_2) de S, le segment $[p_1, p_2]$ est contenu dans S ; l'enveloppe convexe d'un ensemble de points S, notée $\mathcal{C}(S)$, est le plus petit ensemble convexe contenant S. L'enveloppe convexe a de nombreuses applications.



7.2 Méthode

Soit D une demi-droite du plan et O un point quelconque. On définit dans le plan privé de O un ordre total $\leq$ appelé *ordre polaire* relatif à la demi-droite D par : $p \leq q$ si $(\widehat{D, \overrightarrow{Op}}) < (\widehat{D, \overrightarrow{Oq}})$ ou si $(\widehat{D, \overrightarrow{Op}}) = (\widehat{D, \overrightarrow{Oq}})$ et

$Oq \leq Op$, où $(\widehat{D,\vec{u}})$ désigne l'angle formé par la demi-droite D et le vecteur $\vec{u}$ dans l'intervalle $[0, 2\pi]$.



Ordre polaire / D
$(p_1, p_2, p_3, p_4, p_5, p_6, p_7)$

Etant donnés un ensemble de points S et un point $O \notin S$, on définit le *circuit polaire* de S par rapport à O, noté $CP(S)_{/O}$, par une suite (circulaire) des points de S ordonnée par rapport à l'ordre polaire relatif à une demi-droite quelconque d'origine O. On remarque que si $(p_0, \cdots, p_{n-1})$ est un circuit polaire d'un ensemble de points par rapport à un point $O \notin S$, cette liste est ordonnée par rapport à l'ordre polaire relatif à la demi-droite d'origine O et de vecteur directeur $\overrightarrow{Op_0}$.

Etant donné un triplet de points $t = (p_1, p_2, p_3)$, on dit que t est un *tour gauche* (resp. *droit*) si le point p_3 est situé strictement à gauche (resp. à droite) de la droite orientée selon le vecteur $\overrightarrow{p_1p_2}$. On pourra vérifier que t est un tour gauche si le déterminant $\det(\overrightarrow{p_1p_2}, \overrightarrow{p_1p_3})$ est strictement positif.

Le plan étant orienté, le *contour direct* d'un polygone est obtenu en parcourant les arêtes du polygone de sorte que son intérieur soit toujours à gauche.

L'*algorithme de Graham* repose sur deux propriétés très simples :

1. Le contour direct d'un polygone convexe correspond exactement au circuit polaire des sommets de ce polygone par rapport à un point quelconque à l'intérieur de ce polygone[1].
2. Un polygone est convexe si et seulement si tout triplet (p_1, p_2, p_3) de points consécutifs pris sur son contour direct est un tour gauche.

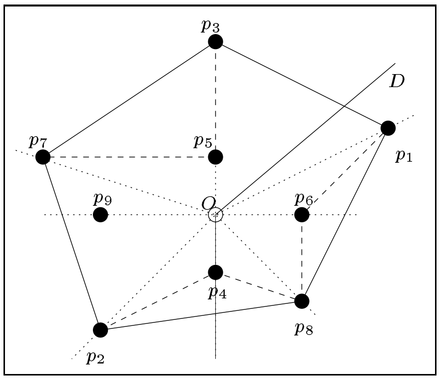
On effectuera tout d'abord un prétraitement de l'ensemble des points S afin d'obtenir une liste ordonnée des points de S qui permettra de calculer le contour (direct) de l'enveloppe convexe de S. On détermine un point

1. cette propriété n'est pas vérifiée pour un polygone quelconque mais n'est pas caractéristique des polygones convexes

$O \notin S$ à l'intérieur de $\mathcal{C}(S)$ (il suffit de prendre le barycentre de trois points non alignés de S qui sera forcément à l'intérieur de $\mathcal{C}(S)$). Il s'agit alors de construire une liste ordonnée $\ell = \langle p_1, p_2, \cdots p_n \rangle$ des points de S correspondant au circuit polaire de S par rapport à O.

Pour obtenir $\mathcal{C}(S)$, il suffit alors de parcourir la liste ℓ en partant d'un point appartenant à $\mathcal{C}(S)$ (on pourra prendre, par exemple, le point d'ordonnée minimale et d'abscisse minimale si il y a plusieurs points d'ordonnée minimale), et d'éliminer tous les points p_i tels que (p_{i-1}, p_i, p_{i+1}) ne soit pas un tour gauche.

Sur l'exemple représenté sur la figure ci-dessous, la liste ordonnée relativement à la demi-droite D d'origine O est $\ell = \langle p_3, p_5, p_7, p_9, p_2, p_4, p_8, p_6, p_1 \rangle$. Le parcours de cette liste s'effectue à partir du point d'ordonnée minimale p_2. Le triplet (p_7, p_9, p_2) est un tour droit, tandis que le triplet (p_1, p_3, p_7) est un tour gauche.



7.3 Résultats demandés

Le programme, qui implémente l'algorithme de Graham devra comporter une interface graphique permettant la saisie des points et l'affichage du résultat.

On n'utilisera pas les fonctions trigonométriques et leurs réciproques, sources d'erreurs d'approximations.

8 Analyse syntaxique**

8.1 Objectif

Réaliser l'analyse syntaxique d'un langage de programmation simple. Les programmes considérés sont de la forme (voisine de CAML) :

```
let
  x1 = 1
and
  x2 = (x1 + 1)*23
in
  2*x1 + x2.
```

8.2 Méthode

Il s'agit d'analyser un texte, c'est-à-dire une suite de caractères. On procède en deux étapes : la première détermine les unités lexicales*, la seconde, les unités syntaxiques**.

Le texte est d'abord découpé en *lexèmes* : dans l'exemple ci-dessus, `23`, `let`, `x2`, `(`, `=` sont des lexèmes, tandis que `let x1`, `x` n'en sont pas. Chaque lexème appartient à une *unité lexicale*, qui comporte un ou plusieurs lexèmes. Voici les unités lexicales correspondant aux exemples précédents : nombre(23) (qui contient tous les lexèmes numériques), let (qui contient seulement `let`), ident(x2) (qui contient tous les identificateurs), par-gauche, egal qui ne contiennent chacune qu'un seul lexème. Certaines unités lexicales sont affectées d'une valeur (par exemple, la valeur numérique d'un nombre, ou la valeur textuelle d'un ident). Par conséquent, les unités lexicales seront représentées en C par un type `struct token`, dont un des champs détermine le type `token_t` de l'unité lexicale, les autres champs servant éventuellement à la valeur lexicale.

L'analyse lexicale d'un texte consiste à le transformer en une suite d'unités lexicales. Par exemple `x2 = (x1 + 1)*3` est transformé en ident(x2) egal par-gauche ident(x1) plus nombre(1) par-droite mult nombre(3).

L'analyse syntaxique travaille sur cette suite d'unités lexicales pour reconnaître des unités syntaxiques, par exemple *programme*, *déf*, *exp*. La suite précédente est reconnue comme une *déf*. Cette reconnaissance se fait à l'aide de règles de grammaire ; l'une d'elle est :

déf:
  ident égal *exp*

Quand ces règles de grammaire sont suffisamment simples, elles peuvent être transcrites automatiquement en définitions de fonctions, mutuellement récursives ; la fonction suivante reconnaît une *déf*, en acceptant un ident, un égal, puis en reconnaissant une *exp* :

```
void def()
{
  match(IDENT); match(EGAL); exp();
}
```

La fonction `match` accepte l'unité lexicale courante ou déclenche une erreur, la variable `lookahead` est globale :

```
void match(token_t t) {
  if (lookahead.type == t)
    lookahead = lexer();
  else
    erreur(...);
}
```

Quand deux règles peuvent s'appliquer à une même unité syntaxique, par exemple

exp:
  terme
  terme plus *exp*

La fonction de reconnaissance de *exp* doit tester l'unité lexicale :

```
void exp() {
  terme();
  if (lookahead.type == PLUS) {
    match(PLUS);
    exp();
  }
}
```

Ces fonctions de reconnaissance ont pour objet de décider si un programme est correct. La reconnaissance des unités syntaxiques peut être accompagnée de l'exécution de certaines opérations, par exemple l'évaluation des expressions.

8.3 Résultats demandés

Les unités lexicales sont les entiers, les identificateurs (commençant par une lettre suivie de lettres ou de chiffres et qui ne sont pas des mots-clés), les parenthèses, le point, les opérateurs d'addition et de multiplication, le signe égal et les mots clés. Le type d'une unité lexicale est représenté par une constante d'un type énuméré `token_t` : `NOMBRE`, `IDENT`, `EGAL`, etc. Les espaces (blancs, nouvelle ligne, etc) doivent être ignorés.

La fonction `token lexer();` retourne à chaque appel une unité lexicale, et avance jusqu'au caractère suivant dans l'entrée standard. Elle n'est appelée que par la fonction `void match(token_t t)`, sauf la première fois.

La grammaire du langage étudié est la suivante :

programme:
  let *définitions* in *exp* point

définitions:
  déf
  déf and *définitions*

déf:
  ident égal *exp*

exp:
  terme
  terme plus *exp*

terme:
  facteur
  facteur mult *terme*

facteur:
  nombre
  ident
  par-gauche *exp* par-droite

À chaque unité syntaxique correspond une fonction de reconnaissance. Ces fonctions opèrent sur une variable globale `lookahead` de type `token`. La fonction `void match(token_t t)` est appelée par les fonctions de reconnaissance.

Dans un premier temps, on réalisera seulement l'analyse, lexicale et syntaxique, d'un programme : dire si un texte est un programme syntaxiquement correct, ou afficher un message d'erreur. On rappelle que l'appel `in.putback(c)` replace sur le flot d'entrée `in` le caractère déjà lu `c` et recule d'une position.

Dans un deuxième temps, on traitera l'évaluation de ces programmes, le résultat étant la valeur de l'expression finale. On devra utiliser une *table des* *identificateurs* qui contiendra la valeur de chaque variable dès qu'elle aura été calculée ; il faudra construire cette table au cours de l'analyse syntaxique (et en particulier modifier la fonction `match` à cette fin). L'indice d'une variable dans la table, ainsi que le lexème correspondant, devront figurer dans l'objet de type `token` retourné par la fonction `lexer()`.

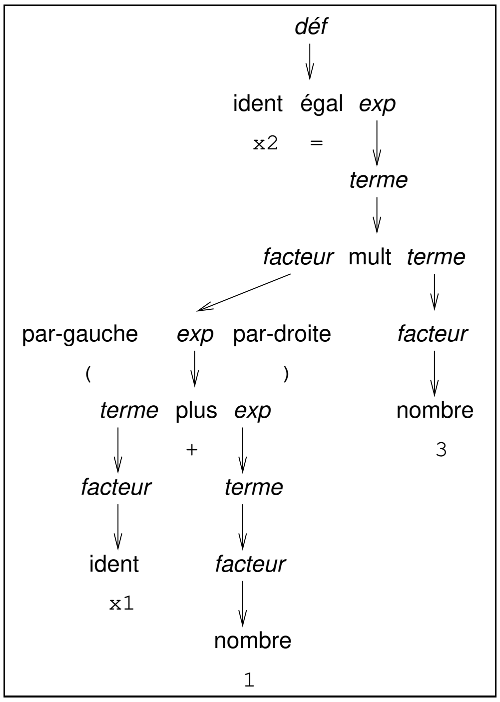


FIG. 1 – *Analyse de* `x2 = (x1 + 1)*3`

9 Jeux de Conway : jeu de Hackenbush***

9.1 Objectif

Un jeu de Hackenbush est défini par un graphe non-orienté planaire connexe dont un nœud particulier est désigné comme la Terre. Dans le jeu bicolore, les arêtes du graphe sont colorées en noir ou en blanc (à peu près du même nombre) ; dans le jeu gris, toutes les arêtes sont grises. Le jeu consiste à retirer des arêtes ; quand une arête est retirée, certains nœuds cessent d'être reliés à la Terre, et on doit aussi retirer ces nœuds et les arêtes ayant ceux-ci pour extrémité. Dans le jeu bicolore, Gauche ne peut choisir qu'une arête blanche et Droit une arête noire ; dans le jeu gris, les joueurs peuvent choisir n'importe quelle arête. Le joueur qui ne peut plus retirer d'arête perd. On représente généralement un jeu de Hackenbush à l'aide d'une figure dans un cadre, celui-ci représentant la Terre.

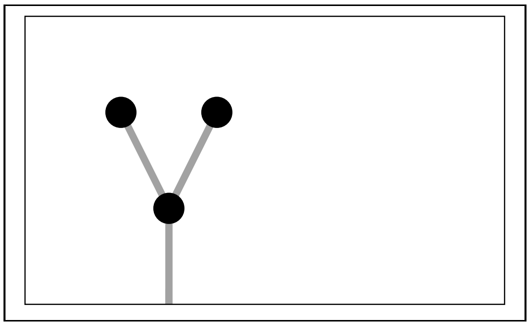


FIG. 2 – *Une figure de Hackenbush*

L'objectif est de programmer un jeu de Hackenbush.

9.2 Principe

Les jeux de Conway sont définis récursivement de la façon suivante : un jeu de Conway est un couple (G, D), où G et D sont deux ensembles finis, éventuellement vides, de jeux de Conway. On notera $(g_1, \ldots, g_p | d_1, \ldots, d_q)$ au lieu de $(\{g_1, \ldots, g_p\}, \{d_1, \ldots, d_q\})$.

Voici quelques exemples, avec leur notation usuelle :

- $\mathbf{0} = (|) = (\emptyset, \emptyset)$
- $\mathbf{1} = (\mathbf{0}|) = (\{\mathbf{0}\}, \emptyset)$
- $\mathbf{-1} = (|\mathbf{0}) = (\emptyset, \{\mathbf{0}\})$
- $\star = (\mathbf{0}|\mathbf{0}) = (\{\mathbf{0}\}, \{\mathbf{0}\})$

On peut dériver de cette définition récursive une représentation arborescente d'un type particulier, où les fils d'un nœud peuvent être de deux sortes : fils gauche ou fils droit.

Les jeux de Conway peuvent être interprétés comme des jeux à deux joueurs, qu'on appelera Gauche et Droit, qui doivent jouer alternativement ; une configuration du jeu est la donnée du joueur qui doit jouer et d'un jeu de Conway.

Si Gauche doit jouer $(g_1, \ldots, g_p | d_1, \ldots, d_q)$, avec $p \neq 0$, il choisit l'un des g_i et ensuite Droit doit jouer g_i ; si $p = 0$, alors Gauche perd. Inversement,

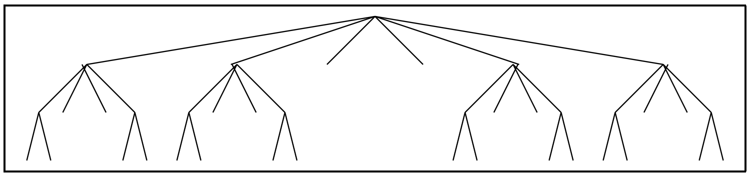


FIG. 3 – *Un arbre de Conway*

si Droit doit jouer $(g_1, \ldots, g_p | d_1, \ldots, d_q)$, avec $q \neq 0$, il choisit l'un des d_i et ensuite Gauche doit jouer d_i ; si $q = 0$, alors Droit perd.

Les configurations perdantes sont donc celles de la forme Gauche joue $(| \ldots)$ ou Droit joue $(\ldots |)$: le perdant est celui qui doit mais ne peut plus jouer.

Comme un nœud de l'arbre détermine deux configurations de jeu, selon que Gauche ou Droit doive jouer, on va affecter à chaque nœud un couple $g \cdot d$ d'attributs, chacun ayant la valeur G (gagnant) ou P (perdant) ; ces attributs sont synthétisés, c'est-à-dire calculées de façon ascendante selon les règles suivantes :

– $(- \cdot G, \ldots, - \cdot G | \ldots) \rightarrow P \cdot -$ (tous les fils gauches ont des attributs de la forme $- \cdot G$)
– $(\ldots, - \cdot P, \ldots | \ldots) \rightarrow G \cdot -$ (au moins un fils gauche a un attribut de la forme $- \cdot P$)
– $(\ldots | G \cdot -, \ldots, G \cdot -) \rightarrow - \cdot P$ (tous les fils droits ont des attributs de la forme $G \cdot -$)
– $(\ldots | \ldots, P \cdot -, \ldots) \rightarrow - \cdot G$ (au moins un fils droit a un attribut de la forme $P \cdot -$)

Selon la valeur de l'attribut à la racine, on a 4 catégories de jeux de Conway :

– $P \cdot P$: le joueur qui commence perd (jeu *nul*)
– $P \cdot G$: le joueur Droit gagne (jeu *négatif*)
– $G \cdot P$: le joueur Gauche gagne (jeu *positif*)
– $G \cdot G$: le joueur qui commence gagne (jeu *flou*)

Les jeux de Conway notés $\mathbf{0}$, $\mathbf{-1}$, $\mathbf{1}$ et $\star$ sont des exemple de jeux de Conway respectivement nul, négatif, positif et flou. Cette classification suppose que chaque joueur applique la stratégie de jeu correspondant au calcul de l'attribut, stratégie qui n'est réalisable que si l'ensemble de l'arbre est connu.

Les jeux de Hackenbush (bicolore ou gris) sont des jeux de Conway. La figure 2 définit un jeu de Conway représenté par l'arbre de la figure 3.

9.3 Résultats demandés

On demande d'écrire un programme qui :

- permette de saisir graphiquement une « figure de Hackenbush »*
- contrôler une partie à deux joueurs (en particulier en supprimant tous les nœuds déconnectés à la Terre)**
- jouer contre l'ordinateur, en choisissant une stratégie***

10 Simulation écologique de stratégies de jeux*

10.1 Objectif

Expérimenter des stratégies du « jeu itéré du prisonnier », qui modélise certains comportements sociaux ou économiques dans le temps pour répondre à la question : y a-t-il intérêt à coopérer ?

10.2 Principe

À chaque tour, les deux joueurs A et B choisissent indépendamment de coopérer (c) ou de trahir (t). Un *coup* est donc l'un des 4 couples (c, c), (c, t), (t, c), (t, t) ; le gain de A est alors respectivement de 6, 0, 5, 2 ; le gain de B est, symétriquement, de 6, 5, 0, 2.

Un *jeu* $\mathbf{x}$ est une suite finie de coups. On note $g_A(\mathbf{x})$ la somme des gains de A pour ces coups, et de même $g_B(\mathbf{x})$ (mais on ne travaillera qu'avec $g_A(\mathbf{x})$, qu'on notera simplement $g(\mathbf{x})$).

Une *stratégie* est une fonction qui à tout entier n et tout jeu $\mathbf{x}$ de n coups ($n \geq 0$), associe c ou t (un demi-coup). Si A adopte une stratégie S, et B une stratégie S', le jeu noté $S \otimes S'$ est alors la suite $(a_0, b_0), (a_1, b_1), (a_2, b_2), \ldots$ où

$$\begin{array}{rcl} a_0 & = & S() \\ b_0 & = & S'() \\ a_1 & = & S((a_0, b_0)) \end{array}$$

$$
\begin{aligned}
b_1 &= S'((a_0, b_0)) \\
a_2 &= S((a_0, b_0), (a_1, b_1)) \\
b_2 &= S'((a_0, b_0), (a_1, b_1)) \\
&\dots \\
a_n &= S((a_0, b_0), \dots, (a_{n-1}, b_{n-1})) \\
b_n &= S'((a_0, b_0), \dots, (a_{n-1}, b_{n-1}))
\end{aligned}
$$

Il y a des stratégies sans mémoire (c'est-à-dire n'utilisant pas les coups précédents), à mémoire bornée (utilisant m coups précédents) ou à mémoire non bornée. Par exemple, on pourra essayer

- les deux stratégies constantes (qui coopérent ou trahissent toujours),
- la stratégie lunatique (qui tire le demi-coup au hasard),
- des stratégies périodiques, sans mémoire (par exemple *cctcct*...),
- la stratégie donnant-donnant (qui commence par coopérer, et joue ensuite comme l'adversaire au coup précédent),
- la stratégie graduelle (qui commence par coopérer, puis quand l'adversaire trahit, réplique par n trahisons successives, où n est le nombre de trahisons passées de l'adversaire, suivies de deux coopérations),
- la stratégie rancunière (qui coopère jusqu'à ce que l'adversaire trahisse, puis trahit toujours),
- la stratégie de majorité (qui commence par coopérer, puis joue comme la majorité des coups de l'adversaire).

On considère p stratégies S_1, ..., S_p et on étudie les p^2 jeux $S_i \otimes S_j$ déterminés par ces stratégies. On peut évaluer une stratégie S_i choisie par A en la confrontant à toutes les autres et en calculant

$$g(S_i) = \sum_{\substack{j=1 \\ j \neq i}}^{p} g(S_i \otimes S_j);$$

selon ce type de confrontation (un *tournoi*), la meilleure sera celle qui maximise $g(S_i)$.

Une autre comparaison des stratégies peut se faire à l'aide d'une *simulation écologique* : on travaille avec une population de q stratégies formée de q_j exemplaires de S_j (avec $q = q_1 + \ldots + q_p$), et on confronte ces q stratégies entre elles. Le gain de S_i confrontée aux $q - 1$ autres est

$$g^{\mathrm{e}}(S_i) = \sum_{j=1}^{p} q_j g(S_i \otimes S_j) - g(S_i \otimes S_i).$$

On part d'une population initiale uniforme, où les q_j sont égaux. À chaque génération, la répartition des q_i est modifiée, à effectif total constant, proportionnellement aux gains de chqque type de stratégie, les nouveaux effectifs des p stratégies étant

$$q'_j = q \frac{q_j g^{\mathrm{e}}(S_j)}{\sum_{i=1}^{p} q_i g^{\mathrm{e}}(S_i)}, \qquad (j = 1, \ldots, p)$$

10.3 Résultats demandés

Il s'agit de programmer un certain nombre de stratégies, classiques ou non, de les classer par évaluation dans un premier temps, puis de faire évoluer une population formée de certaines de ces stratégies par une simulation écologique.

On présentera graphiquement l'évolution des populations en fonction du temps en mettant en évidence des phénomènes intéressants (oscillations, stabilisation, etc).