DEA de Chimie Informatique et Théorique
C, Algorithmique et Programmation

Bruno Mermet

1996-1997

Résumé

Ce document est le support d'un cours d'environ 30 heures de C, Algorithmique et Programmation destiné des élèves en DEA de Chimie Informatique et Théorique. Par conséquent, les points abordés ne sont pas toujours traités à fond, mais le minimum nécessaire pour comprendre les notions présentées est précisé.

Le cours est diviseé en 4 chapitres. Le premier décrit le langage C. Le deuxième aborde quelques notions théoriques de l'informatique. Le troisième chapitre est consacré à la représentation des structures de données en général, avec application en langage C (les fonctions C illustrant les algorithmes ne sont pas exemptes d'erreurs : au lecteur de les trouver ; petite aide : il n'y en a pas dans les paragraphes sur les matrices et sur les graphes). Enfin, le quatrième et dernier chapitre présente un grand nombre d'algorithmes fondamentaux de l'informatique.

Table des matières

Chapitre 1

Le langage C

1.1 Un premier programme

```
/* Le classique programme "Hello World" */
#include <stdio.h>
main()
{
        printf("Hello World \n");
}
```

Les premières notions :

- Le texte entre /* et */ correspond à des commentaires : il n'est pas interprété par la machine ;
- La commande #include sert à *inclure* un fichier. En règle générale, si le nom est entre <>, il s'agit d'un fichier du système. S'il est entre guillemets, il s'agit par contre d'un fichier saisi par l'utilisateur ;
- Le fichier *stdio.h* (*standard input/output header*) est nécessaire pour utiliser les fonctions d'entrée/sortie [1] ;
- Le texte *main()* [2] annonce le programme principal (point de départ de l'exécution du programme). Le programme en lui même est composé de l'ensemble du texte qui suit, et est entre accolades ;
- Toute instruction C se termine par un point-virgule (;) ou une accolade fermante si elle avait débuté par une accolade ouvrante ;
- L'instruction *printf* sert à afficher des messages. Ces messages peuvent être des nombres, des chaînes de caractères [3], ou un mélange des deux ;
- Dans une chaîne de caractères entre guillemets, un caractère précédé d'un *backslash* (\) est un caractère spécial. Ici, le \n (*newline*) signifie que l'ordinateur devra aller à la ligne après avoir écrit le texte entre guillemets.

1.2 Les variables en C

1.2.1 Définition

Une variable possède :

- un nom ;

1. On appelle entrée/sortie tout ce qui a trait aux *périphériques*, liens entre la machine et l'extérieur. Par exemple l'écran, le clavier, le disque dur sont des périphériques.
2. il s'agit d'un en-tête de fonction
3. d'une manière générale, On appelle chaîne de caractères du texte

- un type ;
- une valeur.

Le nom d'une variable est fixe ; c'est ce qui caractérise la variable. De même, le type d'une variable est fixe. Mais sa valeur peut être modifiée.

Dans le nom des variables, on distingue majuscules et minuscules ; donc *toto* et *Toto* sont deux variables différentes. Un nom de variable commence toujours par une lettre. Les autres caractères peuvent être des lettres, des chiffres ou le tiret de soulignement : _. Des noms de variables trop courts manquent souvent de sens, et rendre le programme peu compréhensible. Des noms de variables trop longs rendent le programme illisible[4].

1.2.2 Déclaration

Pour être utilisée, une variable doit être déclarée. Pour déclarer une variable, il faut donner son type, son nom, puis, éventuellement une valeur initiale.

On appelle *type de base* les types déjà définis dans le langage (on verra par la suite que l'on peut aussi définir ses propres types). Les types de bases sont :

- **char** : les caractères (un seul symbole) ;
- **int** : les entiers ;
- **short** : encore des entiers ;
- **long** : toujours des entiers ;
- **float** : les décimaux ;
- **double** : encore des décimaux ;
- **long double** : et toujours des décimaux.

Les variables peuvent être déclarées de deux façons : de manière *globale* ou *locale*. Dans le premier cas elles seront déclarées au début du fichier, dans le deuxième cas, juste après l'accolade ouvrante suivant un en-tête de fonction[5].

Note : Les variables déclarées de manière globale sans initialisation spécifique de la part du programmeur sont initialisées à zéro, tandis que les variables locales ont une valeur indéterminée. Cependant, dans tous les cas, il est conseillé de spécifier à quelle valeur une variable est initialisée.

4. les noms de variables sont, de plus, en général, limités à 255 caractères
5. dans notre cas, après le $main()\{$.

Exemple :

```
#include<stdio.h>
/* Declaration d une variable gobale appelee toto, de type entier. */
int toto;
main()
{
/* Declaration d une variable locale titi, decimale, initialisee a 1,5. */
        float titi = 1.5;

        printf("Hello World\n");
}
```

Note : On peut aussi déclarer les variables de façon multiple si elles ont le même type. Par exemple la séquence d'instructions $int a; int b;$ peut être remplacée par l'instruction $int\ a, b;$.

1.2.3 Utilisation d'une variable

Pour modifier la valeur d'une variable v, il faut utiliser une instruction d'affectation. Ainsi, pour donner à la variable v la valeur 1, il faut taper $v = 1;$.

Pour utiliser la valeur d'une variable, il suffit d'utiliser le nom de la variable. Ainsi, la séquence d'instruction suivante affecte la valeur 1 à v_1 *et* à v_2 :

$$v_1 = 1; v_1 = v_2;$$

Note : Une affectation multiple est possible : l'instruction $a = b = c = d$ recopie dans a, b et c la valeur de d.

1.2.4 Affichage de la valeur d'une variable

L'affichage de la valeur d'une variable se fait grâce à la fonction printf.

Exemple :

```
#include<stdio.h>
main()
{
        float approx_pi = 22./7.;

        fprintf("le nombre %d est une valeur approchee de π.\n", approx_pi);
}
```

Le caractère % signifie que la valeur d'une variable, parmi celle qui suivent (à l'intérieur des parenthèses, séparées par des virgules) doit être affichée à cet endroit. La lettre 'd' qui suit le symbole % indique que c'est une variable de type entier.

On, peut, dans une même instruction *printf*, afficher la valeur de plusieurs variables. Auquel cas les variables sont affichées dans l'ordre dans lequel elles sont citées.

Exemple : Le programme suivant :

```
#include<stdio.h>
main()
{
        int a, b, r ;

        b = 22;
        a = 7;
        r = b / a;
        fprintf("la division euclidienne %d/%d a pour resultat %d.\n", a, b, r);
}
```

Donne le résultat :

la division euclidienne 22/7 a pour resultat 3.

Note : En fait, *printf* n'affiche pas des variables, mais des *expressions*, dont les variables ne sont qu'une forme particulière.

1.3 Affichage des expressions : introduction

Toute expression possède un type. De même, les opérateurs sont définis pour des types bien particuliers. Mais un grand nombre de conversions sont faites automatiquement pour rendre ces types compatibles. Il est **fortement déconseillé** d'abuser de ces conversions automatiques.

Voici résumé de quel caractère il faut faire suivre le symbole %[6] pour afficher

6. pour faire afficher un %, il faut mettre %%

une expression d'un type donné.

type	*symbole*
char	*c*
int	*d*
short	*hd*
long	*ld*
float	*e, f ou g*
double	*le, lf ou lg*
longdouble	*Le, Lf ou Lg*

Note : pour les nombres décimaux, les résultats, suivant le symbole utilisé, sont les suivants :

- f : forme standard ;
- e : forme scientifique ;
- g : la mieux adaptée des deux.

1.4 Instructions de base

1.4.1 Expressions

Deux types d'instruction ont déjà été vues : l'instruction d'affichage *printf* et l'instruction d'affectation =. La forme générale de cette dernière est *variable* = *expression*. Une expression est assimilable, en règle générale, à une formule mathématique. Voici quelques exemples d'expressions :

expression	*valeur*	*type*
$2 + 3$	5	*entier*
$2.1 + 3.$	5.1	*decimal*
$3./2$	1.5	*decimal*
$(2 == 3)$	0	*entier*
$(2 != 3)$	1	*entier*
$(2 == 3 \|\| 2 == 2)$	1	*entier*

opérateurs de base principaux

Remarque : les opérateurs sur les bits notamment, ne sont pas décrits.

symbole	*exemple*	*valeur*	*role*
		symboles arithmetiques	
+	$2+3$	5	*addition des entiers et decimaux*
−	$2-3$	-1	*soustraction des entiers et decimaux*
*	$2-3$	6	*multiplication des entiers et decimaux*
/	$2/3$	0	*division des entiers et decimaux*
%	$2\%3$	1	*reste de la division entiere*
		symboles logiques	
==	$2==3$	0	*test d'egalite*
!=	$2!=3$	1	*test de difference*
<	$3<2$	0	*test d'inferiorite stricte*
>	$3>2$	1	*test de superiorite stricte*
<=	$3<=3$	1	*test d'inferiorite au sens large*
>=	$3>=2$	1	*test de superiorite au sens large*
!	!1	0	*negation*
||	0||1	1	*le "ou" logique (inclusif)*
&&	0&&1	0	*le "et" logique*

Note : Il existe des formes abrégées des affectations utilisant certains de ces symboles. Ainsi, l'affectation :

$$x = x - 4;$$

peut s'écrire plus simplement :

$$x-=4;$$

Une forme encore plus simple existe lorsqu'une variable est incrémentée ou décrémentée : plutôt que d'écrire

$$x+=1;/*\ raccourci\ de\ x=x+1\ */$$

on peut écrire, au choix, $x{+}{+}$ ou ${+}{+}x$

Note : une affectation est aussi une expression, qui a pour valeur le résultat de l'affectation. C'est la que se situe la différence entre $x{+}{+}$ et ${+}{+}x$ Dans le premier cas, l'incrémentation est faite *après* l'évaluation de x, alors qu'elle est faite *avant* dans le deuxième cas. Ainsi, si x vaut 2, l'instruction $a = 4 + x{+}{+}$ donne à a la valeur 6, et à x la valeur 3, tandis que l'instruction $a = 4 + {+}{+}x$, si elle donne toujours à x la valeur 3, donne par contre à a la valeur 7.

Fonctions mathématiques plus évoluées

Certaines fonctions mathématiques nécessitent, pour être utilisées, d'inclure le fichier *math.h*. C'est notamment le cas des fonctions suivantes :

fonction	*role*
$sin(x), cos(x), tan(x)$	*sinus, cosinus et tangente de* x*(*x *en radian)*
$asin(x), acos(x), atan(x)$	*arcsinus, arccosinus et arctan de* x
$atan2(y, x)$	$arctan(y/x)$ *entre* $-\pi$ *et* π
$sinh(x), cosh(x), tanh(x)$	*sinus, cosinus et tangente hyperbolique*
$exp(x), log(x), log10(x)$	*exponentielle, logarithme neperien et* log_{10}
$pow(x, y), sqrt(x)$	$x^y, \sqrt{y}$
$ceil(x)$	$min\{y \vert y \in \mathbb{Z} \text{ et } y \geq x\}$
$floor(x)$	$max\{y \vert y \in \mathbb{Z} \text{ et } y \leq x\} = \vert x \vert$
$fabs(x)$	*partie de* x *avant la virgule*

1.4.2 La boucle *for*

Il s'agit d'une instruction qui permet de répéter un certain nombre de fois une instruction donnée [7]

Cette instruction a classiquement la forme suivante :

for(*init*;*cond*;*evol*)
    inst

où :

- *init* est une instruction d'initialisation (par exemple, $i = 0$) [8] ;
- *cond* est une condition (expression) de continuation, faisant souvent référence au compteur : on répète *inst* tant que cette condition est vraie (par exemple $i <= 10$) ;
- *evol* est une instruction qui fait évoluer le compteur, et est donc susceptible de modifier la valeur de *cond* ;
- *inst* est une instruction.

Note : le test *cond* est effectué *avant* d'exécuter le corps de la boucle. Aussi, si celui est faux avant la première itération, l'instruction *inst* n'est pas exécutée.

Exemple : calcul de la somme des entiers de 0 à 100

7. Rappel : une suite d'instructions entre accolades est une instruction
8. par la suite, on désignera la variable i par le terme *compteur*

```
#include<stdio.h>
/* Programme qui calcule la somme des entiers de 0 a 100 */
main()
{
        int somme = 0; /* on initialise la somme a zero */
        int compteur; /* la variable qui servira de compteur dans la boucle */

        for(compteur=0 ; compteur <= 100 ; compteur++)
                somme +=compteur;
/* on rajoute a la somme des entiers de 0 a compteur-1 la valeur compteur */
/* ce qui donne la somme des entiers de 0 a compteur */
        printf("La somme des entiers de 0 a 100 vaut %d\n", somme);
}
```

1.4.3 La boucle *while*

Il s'agit d'une autre forme de boucle :

while (*cond*)
    inst

La condition *cond* est vérifiée *avant* chaque itération. Si elle est évaluée à faux, on sort de la boucle.

1.4.4 La boucle *do-while*

Et encore un type de boucle :

do
    inst
while (*cond*)

Par rapport à la boucle *for* ou à la boucle *do-while*, le test est effectué *après* chaque itération. Donc s'il est faux au moment de rentrer pour la première fois dans le corps de la boucle, celui-ci est quand même exécuté.

1.4.5 L'instruction *if*

Cette instruction permet, en fonction d'une condition d'exécuter une instruction ou une autre. Elle existe sous deux formes :

première forme :

if (*cond*)
    $inst_1$

deuxième forme :

if (*cond*)
    $inst_1$
else
    $inst_2$

Dans les deux cas, si la condition *cond* est vérifiée, on exécute $inst_1$. Si elle n'est pas vérifiée, dans le premier cas, on ne fait rien, tandis que dans le deuxième cas, on exécute l'instruction $inst_2$.

Exemple :

```
#include<stdio.h>
/* Programme qui demande a l'utilisateur de taper un caractere.
Celui-ci a trois essais. Si le caractere tape est un '0' on
affiche "gagne", et on arrete. Si au bout de 3 essais, l'utilisateur
n'a toujours pas tape de "1", on sort en affichant "perdu".*/
main()
{
        char c = '\0'; /* Caractere tape par l'utilisateur, initialise a
la valeur "caractere nul" (attention, ne pas confondre avec '0') */
        int numero_essai /* numero de l'essai en cours

        for(numero_essai = 1; numero_essai <= 3 && c != '0'; numero_essai++)
                c= getchar(); /* lecture d'un caractere au clavier */
        if (c == '0')
                printf("Gagne\n");
        else
                printf("Perdu\n"); }
```

Note : "gag" classique du débutant : taper (c = '0') comme condition du *if* au lieu de (c == '0'). C'est syntaxiquement correct, mais le résultat n'est pas exactement celui escompté...

1.4.6 Le choix *switch*

Lorsque la valeur d'une variable par rapport à des constantes détermine des exécutions de différentes instructions, plutôt que d'utiliser la lourde structure des *else if*, on peut utiliser l'instruction *switch*.

Exemple :

```
switch(c)
        {
                case 0 :
                        inst1 ;
                        break;
                case 2 :
                case 4 :
                        inst2 ;
                case 5 :
                        inst3 ;
                        break;
                default :
                        inst4 ;
                        break;
        }
```

La sémantique opérationnelle de cet exemple est, suivant les valeurs de c :

- c = 0 : on exécute juste $inst_1$;
- c = 2 ou c = 4 : on exécute $inst_2 ; inst_3$;
- c = 5 : on exécute juste $inst_3$
- c a une autre valeur : on exécute $inst_4$.

1.5 Les tableaux

1.5.1 Les tableaux simples

Les tableaux peuvent être vus comme des types secondaires : ils sont construits sur les types de bases, et ont une structure voisine de celle des matrices mathématiques à une dimension. Leur taille est fixe, et leurs éléments sont tous de même type.

Un tableau se déclare de la façon suivante :

type nom_variable[*taille*]

où :

- *type* désigne un type de base ;

- *nom* est un nom de variable ;
- *taille* est un entier qui donne le nombre d'éléments du tableau. Attention : le premier élément a pour indice 0 !

Pour accéder au n-ième élément d'un tableau nommé *toto*, il faut utiliser la syntaxe suivante : $toto[n-1]$.

Exemple : calcul et stockage des 20 premiers termes de la suite de fibonnacci.

```
#include<stdio.h>
/* suite de Fibonnacci */
main()
{
        int fibo[20]; /* le tableau resultat */
        int i; /* le compteur... */

        fibo[0] = 1; fibo[1] = 1; /* initialisation de la recurrence */
        for(i=2 ; i <=19 ; i++)
                fibo[i] = fibo[i-2]+fibo[i-1];

/* on affiche... mais a l'envers ! */
        for( i = 19 ; i <= 0; i--)
                printf("fibo(%d) = %d\n", i, fibo[i]);
}
```

1.5.2 Les tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions proprement dits n'existent pas en C. Par contre, on peut définir des tableaux de tableaux. L'accès se fait de manière analogue aux tableaux de base.

Exemple :

```
int matrice[4][5]; /* definition d'un tableau de 4 par 5 */
matrice[3][2] = 4 ; /* on fixe a 4 l'element de coordonnees (3,2) */
```

Note : erreur classique du débutant (surtout s'il a déjà utilisé d'autres langages de programmation), accéder ainsi à un élément d'un tableau à deux dimensions : $matrice[3,2]$. C'est syntaxiquement correct (deux expressions séparées par une virgule constituent une expression dont la valeur est celle de la dernière sous-expression), mais ce n'est pas ce que l'on veut. Ici, par exemple cela donne la valeur du *pointeur* sur la troisième ligne du tableau *matrice*.

1.6 Les structures

Il peut être intéressant quelquesfois[9] d'associer plusieurs caractéristiques à une variable. Ainsi, un point d'un plan[10] est défini par une abscisse et une ordonnée. Associer ces deux caractéristiques pour définir un nouveau type peut être réalisé au moyen du mot-clef *struct*. On définit un tel nouveau type ainsi :

```
struct pt
{
        float x;
        float y;
};
```

Par la suite, une variable pourra être déclarée de ce type ainsi :

```
struct pt origine;
```

Et l'accès (lecture et écriture) aux différents champs se fera ainsi :

```
origine.x = 0; /* ecriture */
origine.y = 0; /* ecriture encore*/
origine.x+origine.y /* lecture */
```

Comme taper *struct pt* pour chaque nouveau point n'est pas agréable, on peut utiliser le mot-clef *typedef* ainsi (on suppose la structure pt déjà définie) :

```
typedef struct pt point;
```

et par la suite, déclarer ainsi les variables :

```
point origine;
```

Note : l'habitude veut que ces deux étapes soient faites en même temps. On écrira donc plutôt :

```
typedef struct pt
{
        float x;
        float y;
} point;
```

1.7 Les unions

Une variable peut, dans certains cas, avoir plusieurs types différents, selon le cas dans lequel on est, sans pourtant avoir en même temps deux types différents. Une telle propriété est implantable grâce à la notion d'union :

9. c'est ce qu'on appelle un euphémisme
10. désolé pour la banalité de l'exemple

```
union type_u
{
        int i;
        float x;
};

type_u var;
```

Après les déclarations suivantes, si l'on sais que *var* représente un entier, on accédera à sa valeur par un *var.i*, alors que s'il représente un décimal, on utilisera *var.x*. Cependant, attention : l'affectation d'une valeur d'un type donné écrase la valeur précedente de *var*, même si elle était d'un autre type. Qui plus est, il ne faut surtout pas croire que si l'on fait $var.i = 3$, alor $var.x$ vaudra 3. !

1.8 Les types énumérés

Il s'agit d'une manière pratique de définir un ensemble fini de valeurs pour une variable, par exemple un ensemble d'états.

Exemple :

```
enum jour {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche} ;
```

On peut alors par la suite déclarer une variable de type *jour*, et lui donner des valeurs de la liste associé à ce type.

Exemple :

```
jour jj ;

jj = mardi;
```

> Note : l'utilisation de *enum* est analogue à l'utilisation de la commande *#define* (voir paragraphe 1.17. Cependant, elle est plus claire, et un contrôle de type peut lui être associé.

Dans le cas d'une énumération, le premier élément se voit automatiquement affecté la valeur zéro. Cependant, en cas de nécessité, cela peut être changé. Exemple :

```
enum jour {lundi = 1, mardi, mercredi, jeudi, vendredi,
```

samedi, dimanche} ;

On peut aussi avoir des valeurs non séquentielles :

```
enum jour {lundi = 'L', mardi = 'M', mercredi = 'm',
        jeudi = 'J', vendredi = 'V', samedi = 'S', dimanche = 'D'} ;
```

1.9 Les pointeurs

1.9.1 Introduction : les chaînes de caractères

On a déjà vu que les chaînes de caractères existaient (heureusement d'ailleurs), or aucun type ne semble leur être associé. Pourquoi ? tout simplement parce qu'une chaîne de caractères, ce n'est finalement rien d'autre qu'un tableau de caractères ! Ainsi, on peut définir la variable *chaine* comme cela :

```
char chaine[ ] = "petit texte";
```

Note : toute chaîne de caractères se termine par le caractère nul '\0'.

Ainsi, *chaine*[0] désignera par exemple le caractère 'p', *chaine*[1] le caractère 'e', etc. Mais que désigne *chaine* ? Il s'agit de l'adresse, dans la mémoire de l'ordinateur, à laquelle est stocké le premier élément du tableau. On appelle cela un *pointeur* sur *chaine*[0].

1.9.2 Généralisation de la notion de pointeur

Toute variable étant stockée quelquepart en mémoire, on peut associer un pointeur à toute variable. Par exemple un pointeur sur une variable ainsi déclarée *int i*; peut être obtenu en tapant &*i*, et il sera de type (*int* ∗). On peut aussi déclarer des variables de type pointeur. Inversement, pour obtenir la valeur d'une variable dont *j* est un pointeur, on utilisera ∗*j*

Le type *pointeur* est considéré comme un type de base.

Exemple :

```
#include<stdio.h>
/* exemple totalement inutile */
main()
```

```
{
        int i; /* une variable de type entier... */
        (int *) j; /* et une variable de type pointeur sur entier... */

        i = 1;
        j = &i;
        *j = 2;
        printf("i vaut : %d\n", i);

}
```

1.9.3 Arithmétique des pointeurs

On a vu que l'accès au premier élément d'un tableau toto pouvait se faire par $toto[0]$, mais aussi par $*toto$, que l'on peut aussi noter $*(toto + 0)$. Ce principe peut être généralisé à tous les éléments d'un tableau ; par exemple, $*(toto + 2)$ permet d'accéder au troisième élément du tableau. Par extension aux tableaux à plusieurs dimensions, si on a défini le tableau $int\ titi[4][2]$, alors $titi[1]$ est un pointeur sur la deuxième ligne de la matrice $titi$, qui est un tableau de 3 entiers.

Note : S'il s'agissait de l'addition classique, cela voudrait dire que tous les éléments d'un tableau prennent la même place en mémoire (un octet en l'occurence). Or ce n'est assurément pas le cas. En fait, l'addition (ou la soustraction) d'un entier à un pointeur est fonction de la taille du type sur lequel est défini le pointeur. Ainsi, si les entier sont stockés sur 4 octets, et que $toto$ est un pointeur sur entier, l'expression$(toto+2)$ ajoute 8 (2×4) au pointeur toto.

1.9.4 Application : tableaux avant-après

Il peut arriver (cela est même fréquent dans des problèmes de simulation) que l'on ait à faire évoluer un tableau en fonction de son état précédent, et cela plusieurs fois. Une première idée est d'avoir deux tableaux : un tableau $avant$, et un tableau $apres$. On calcule une itération, les nouvelles valeurs se trouvent alors dans le tableau $apres$, qui doit devenir le tableau $avant$ de l'iteration suivante. Il faut donc le recopier. Mais cette étape peut prendre du temps. Or un simple echange de pointeurs peut suffire :

```
main()
{
        int tableau1[10], tableau2[10]; /* les tableaux en memoire */
```

```
        (int *) avant, apres, echange;
/* les tableaux manipules */
        int i, j; /* compteurs */

        /* on initialise les donnees */
        for (i = 0; i <= 9; i++)
tableau[1] = 100 * i;

        avant = tableau1;
        apres = tableau2;

        /* Le calcul se fait en 5 etapes : */
        for(j = 1 ; j <= 5; j++)
                {
                        /* calcul d'une iteration */
                        for(i = 0; i <= 9; i++)
                                apres[i]=avant[i]-avant[i-1];
                        /* preparation iteration suivante */
                        echange = avant;
                        avant = apres;
                        apres = echange;
                }
        /* Attention : le resultat est dans le tableau avant */
}
```

1.9.5 Pointeurs sur structure

On peut aussi définir des pointeurs sur les nouveaux types structurés introduits. Auquel cas l'accès aux champs internes de la structure peut être réalisé de deux manières :

```
        typedef struct pt
        {
                float x;
                float y;
        } point;

        point sommetA ;
        (point *) A_ptr;

        (*A_ptr).x = 3;/* premier type d'acces, classique */
        A_ptr->y = 2; /* deuxieme type d'acces, nouveau */
```

1.10 L'instruction *printf* en détail

1.10.1 forme générale

On a déjà vu le fonctionnement de base de l'instruction *printf*. Cependant, celle-ci permet aussi de formater un peu plus les écritures des expressions. Ainsi, le classique %d peut en fait prendre la forme suivante :

$$\%d\ ta\ p\ ty$$

où :

- *d* est un drapeau qui peut être :
  - # si l'on veut un "." décimal toujours présent ;
  - + si l'on veut un signe *plus* affiché pour les nombres positifs ;
  - *espace*, si l'on veut qu'à l'affichage, un nombre positif soit précédé d'un zéro ;
  - *0* si l'on veut que les nombres de longueur inférieure à *ta* soit précédés de zéros pour compléter à la bonne largeur.
- *ta* est la taille minimum prise par l'argument, c'est-à-dire le nombre minimum de caractères à afficher. Si le nombre donné est positif, l'alignement à l'affichage se fait sur la colonne de droite, s'il est négatif, sur la colonne de gauche. Ce caractère peut être l'astérisque au lieu d'un nombre. Auquel cas c'est l'argument suivant de la fonction *printf* qui détermine la taille ;
- *p* détermine la précision (nombre de chiffre après la virgule) avec laquelle se fait l'affichage. Même rôle de l'astérisque que pour le cas précédent ;
- *ty* est l'identificateur de type déjà vu. On notera cependant que pour les entiers, on peut remplacer *d* par *o* ou *x*[11] si l'on veut un résultat en octal ou en hexadécimal.

1.10.2 écrire des chaînes de caractères

On a vu que l'on pouvait déclarer des chaînes de caractères comme des tableaux de caractères. Mais l'affichage de telles chaînes avec ce que l'on connaît maintenant ne serait pas aisé (obligation de faire un boucle pour afficher la chaîne caractère par caractère). Heureusement, une autre possibilité existe : donner comme après le caractère % le type *s*, qui sera associé à un pointeur sur caractère.

Exemple :

11. ou *X* si l'on préfère

```
char chaine[] = "petit texte";

printf("Voici le texte de la variable chaine : %s\n", chaine);
```

1.11 Demander des informations à l'utilisateur

1.11.1 L'instruction *scanf*

L'instruction *scanf* permet de lire ce que l'utilisateur tape au clavier, et d'affecter tout ou partie de la saisie à une ou plusieurs variable. Sont utilisation est fort voisine de celle du *printf*. Il y a cependant quelques différences. Notamment, les paramètres sont en général des *pointeurs* sur les variables.

Exemple :

```
int i, j ;
char c[5];

scanf("%d %4s %d", &i, c, &j);
printf("vous avez ecrit : %d, %s et %d\n", i, c, j);
```

Si l'on rentre 425 bonjour 326, le programme affiche :

Vous avez ecrit 425, bonj et 326

La forme générale de l'instruction *scanf* est la suivante :

$$\%\, d\ ta\ ty$$

où :

- *d* est un drapeau qui peut être :
  - * si l'on veut que le texte reconnu ne soit affecté à aucune variable.
- *ta* est le nombre de caractères à lire ;
- *ty* est l'identificateur de type déjà vu. De plus, on peut avoir un type de la forme [*texte*], qui permet, dans le cas d'une chaîne de caractères de déterminer exactement les caractères désirés (on s'arrête au premier caractère non désiré).

Exemple :

```
char c[50];

scanf("%[0123456789], c);
printf("vous avez ecrit : %s\n", c);
```

Si l'on tape *234abc32*, le résultat sera :

Vous avez ecrit 234

Avec les crochets, on peut taper des séquences analogues aux suivantes :

$[xyz]$	on accepte des x, des y et des z
$[\hat{}xyz]$	on accepte tout sauf x, y et z
$[a-z]$	on accepte les minuscules seulement
$[]xyz]$	on accepte le crochet droit, x, y et z
$[-xyz]$	on accepte le tiret, x, y et z

Note : le crochet droit doit être le premier caractère entre les crochets (éventuellement précédé du $\hat{}$), et le tiret doit être en première ou dernière position.

1.11.2 La fonction *getchar*

Cette fonction[12] permet aussi de lire une frappe au clavier. Cependant elle ne lit qu'un seul caractère, et celui-ci doit être suivi de la touche *return*. L'utilisation de cette fonction est simple :

```
char c;

c = getchar();
printf("vous avez tape un %c \n", c);
```

12. nous verrons bientôt ce qu'est une fonction

1.12 Les fonctions

1.12.1 Notion de fonction

En mathématiques, une fonction est *quelquechose* défini par un nom, qui prend des arguments (ou paramètres) et rend un résultat. Il en est (presque) de même en C. Cependant, on doit aussi spécifier le type des paramètres et du résultat. Le résultat est renvoyé grâce au mot-clef "return".

Exemple : fonction *plus* sur les entiers

```
int plus(int x, int y)
{
        return x + y;;
}

main()
{
        int a = 3;
        int b = 2;

        printf("La somme de %d et %d vaut %d \n", a, b, plus(a, b));
}
```

Ce petit programme peut se traduire ainsi :

plus est la fonction qui à x (de type entier) et y (de type entier) associe x+y (de type entier). De plus, si $a = 3$ et $b = 2$, alors $f(a, b) = f(3, 2) = 3 + 2$.

Aussi, le résultat du programme est :

```
La somme de 3 et 2 vaut 5
```

1.12.2 Déclaration et utilisation d'une fonction

Une fonction se déclare à l'extérieur de toute autre fonction (y compris le programme *main()*. De plus, elle doit être déclarée avant d'être utilisée. Enfin, une fonction peut ne pas prendre de paramètres, et peut ne pas renvoyer de résultat. Voici les différentes formes d'en-têtes de fonctions suivant ces différents cas :

fonction avec paramètres et résultat :

```
type_retour nom_fonction(liste_parametres)
```

fonction avec paramètres sans résultat :

```
void nom_fonction(liste_parametres)
```

fonction sans paramètre et avec résultat :

```
type_retour nom_fonction(void)
```

fonction sans paramètre et sans résultat :

void nom_fonction(void)

Note : une fonction, comme une variable, est identifiée par son nom. Une fonction ne peut être déclarée plus d'une fois dans un programme.

Dans le cas général, une fonction est appelée ainsi :

- Si la fonction renvoie un résultat d'un type t, alors on affecte ce résultat à une variable de type t par l'intermédiaire d'une instruction de la forme: $var = f(x)$; ;
- Si la fonction ne renvoie pas de résultat, on appelle directement la fonction : $f(x)$; ;

Note : les paramètres et le résultat des fonctions doivent être des types de base. On ne peut donc notamment pas passer en paramètres des structures ou des tableaux.

1.12.3 passage par valeur, passage par adresse

Dans les exemples précédents, on évalue la valeur du paramètre avant de le passer à la fonction. Ce que l'on passe est donc sa *valeur*. Aussi parle-t-on de *passage par valeur*. Mais, le type *pointeur* étant un type de base, on peut aussi passer à une fonction non pas la variable elle-même, mais un pointeur dessus.Ce que l'on passe est donc bien une valeur (celle du pointeur), mais par rapport à la variable qui nous intéresse réellement, il s'agit de l'adresse de celle-ci. Aussi, dans ce cas parle-t-on de *passage par adresse*.

Exemple :

```
int plus_adr((int *)x, (int *)y)
{
        return *x + *y;
}

main()
{
        int a = 3;
        int b = 2;

        printf("La somme de %d et %d vaut %d \n", a, b, plus_adr(&a, &b));
}
```

Ce nouveau programme fait exactement la même chose que le précédent. Mais avec un passage des paramètres par adresse et non plus par valeur. Bien sûr,

sur cet exemple, l'intérêt du passage par adresse n'est pas flagrant. Cependant, on a dit plus haut que l'on ne pouvait passer que des types de base (donc ni structure, ni tableau). Or un pointeur est un type de base. Donc on peut passer à une fonction un pointeur sur structure ou un pointeur sur tableau. ce qui permet de faire manipuler à une fonction ce genre de données.

Exemple :

```
typedef struct pt
{
        float x;
        float y;
} point;

float norme((point *) p1, (point *) p2)
{
        float delta_x;
        float delta_y;

        delta_x = p2->x - p1->x;
        delta_y = p2->y - p1->y;
        return sqrt(delta_x*delta_x + delta_y*delta_y); }

main()
{
        point pa, pb;

        pa.x = 2; pa.y = 4;
        pb.x = 5; pb.y = 8;
        printf("La norme du vecteur ((%g, %g), (%g, %g)) vaut : %g \n",
                pa.x, pa.y, pb.x, pb.y, norme(&pa, &pb));
}
```

L'exécution de ce programme produit :

La norme du vecteur ((2., 4.), (5., 8.)) vaut : 5.

1.12.4 Conséquences du passage par adresse

Dans le cas d'un passage par valeur, une fonction ne peut pas modifier la valeur d'un paramètre (une fois dans la fonction, il n'y a plus aucun lien entre le paramètre et son origine). Par contre, dans le cas d'un passage par adresse...

Exemple : translater un point d'un vecteur donné

```
typedef struct pt
{
        float x;
        float y;
} point;

typedef struct vr
{
        float x;
        float y;
} vecteur;

void translater((point *) p, (vecteur *) v)
{
        p->x+=v->x;
        p->y+=v->y;
}

main()
{
        point pa;
        vecteur va;

        pa.x = 2; pa.y = 4;
        va.x = 1; va.y = 1;
        printf("Le point a se situe en (%g, %g) \n", pa.x, pa.y);
        translation(&pa, &va);
        printf("Le point a se situe en (%g, %g) \n", pa.x, pa.y);
}
```

Résultat de l'exécution :

```
Le point a se situe en (2., 4.)
Le point a se situe en (3., 5.)
```

1.12.5 Portée des identificateurs

Comme cela avait été mentionnée dans la partie consacrée aux variables, et comme on l'a vu pour la fonction *norme*, on peut déclarer des variables locales au début du corps des fonctions C. Entre deux fonctions différentes, des variables locales, même si elles ont le même nom, n'ont aucun rapport. Si une variable locale d'une fonction a le même nom qu'une variable globale, alors dans toute la

définition de la fonction, c'est la variable locale qui est considérée. Elle n'écrase pas la variable globale, mais la *masque*. Autrement dit, hors de la fonction, la variable globale continue à avoir sa signification globale, et l'exécution de la fonction ne l'a *a priori*[13] pas modifiée.

> Note : une variable locale est détruite au sortir de la fonction dans laquelle elle est déclarée. Donc retourner un pointeur sur une variable locale n'a pas de sens : celui-ci pointera sur rien.

1.13 Récursivité ?

1.13.1 Programmer une suite récurrente

Il arrive, en mathématiques, que des suites soient définies de la manière suivante :

$$\begin{cases} U_0 = init \\ U_n = f(U_{n-1}) \end{cases}$$

Exemple : la fonction factorielle :

$$\begin{cases} f(0) = 1 \\ f(n) = n * f(n-1) \end{cases}$$

Ce que l'on peut traduire en bon français (?) par : " f(n) = (si n = 0 alors 1 sinon n * f(n-1))".

De même cela peut se traduire en bon C (!) par :

```
int factorielle(int n)
{
        return (n == 0) ? 1 : n*factorielle(n-1);
}
```

Dans cette nouvelle fonction *factorielle*, on constate que la fonction s'appelle elle-même. Ceci est effectivement *a priori* possible, dans la mesure où la fonction factorielle est déclarée avant son utilisation (c'est l'en-tête d'une fonction qui la déclare). Mais que se passe-t-il lorsque la fonction est appelée ?

13. avec les pointeurs, rien n'est impossible...

exemple : calcul de factorielle(3) :

$$
\begin{array}{ll}
factorielle(3) & = \\
(3 == 0)?1 : 3 * factorielle(2) & = \\
3 * factorielle(2) & = \\
3 * ((2 == 0)?1 : 2 * factorielle(1)) & = \\
3 * (2 * factorielle(1)) & = \\
3 * (2 * ((1 == 0)?1 : 1 * factorielle(0))) & = \\
3 * (2 * (1 * factorielle(0))) & = \\
3 * (2 * (1 * ((0 == 0)?1 : factorielle(-1)))) & = \\
3 * (2 * (1 * (1))) & = \\
3 * (2 * (1)) & = \\
3 * (2) & = \\
6 &
\end{array}
$$

1.13.2 fonction récursive

Une fonction telle que la fonction factorielle (fonction qui s'appelle elle-même) est appelé fonction *récursive*.

Pour le calcul d'une fonction telle que la fonction factorielle, la récursivité n'est pas nécessaire, ni même conseillée, dans la mesure où l'on peut écrire plus simplement cette fonction[14] :

```
int factorielle(int n)
{
        int i;
        int f = 1;

        for(i = 2 ; i ¡= n; i++)
                f *= i; }
```

Cependant, certains problèmes ne peuvent être que très difficilement résolus sans faire appel à la récursivité. C'est notamment le cas (classique) de la résolution du problème des tours de Hanoï.

1.13.3 type récursif

On désire représenter un arbre généalogique. Chaque personne est caractérisée par son nom, son prénom, son père et sa mère. Or les parents d'une personne sont aussi des personne. On aimerait donc pouvoir disposer du nouveau type *personne* suivant :

14. En règle générale, une fonction récursive est plus lente qu'une fonction itérative qui fait le même travail

```
struct pers
{
        char nom[20];
        char prenom[20];
        struct pers pere;
        struct pers mere;
}

struct pers moi;
```

Il se trouve que la définition d'un tel type n'est pas possible. En effet, pour un tel programme, il faudrait réserver pour la variable *moi* la place nécessaire pour le nom, le prénom, le père et la mère. Or si la taille du nom et du prénom est connue, que vaut celle du père? Comme c'est une personne, il lui faut la place pour le nom, le prenom, son père (grand-père de *moi*, sa mère, etc. Donc il faudrait réserver une place mémoire infinie. Comment s'en sortir? Une fois de plus... grâce aux pointeurs :

```
struct pers
{
        char nom[20];
        char prenom[20];
        (struct pers *) pere;
        (struct pers *) mere;
}

struct pers moi;
```

Pourquoi cette version-là est-elle correcte? Comme cela a déjà été dit, lorsque l'on déclare une variable de type *pointeur sur quelquechose*, seule la place nécessaire au stockage du pointeur est réservée. Et celle-ci est fixe (4 octets en général). Donc le compilateur peut réserver pour *moi* une place de $20 + 20 + 4 + 4 = 48$ octets. Le code suivant permet alors de donner à *moi* des parents déclares :

```
struct pers mon_pere;
struct pers ma_mere;
```

```
moi.pere = &mon_pere;
moi.mere = &ma_mere;
```

1.14 Gestion dynamique de la mémoire

1.14.1 Introduction

Jusqu'à présent, quand on déclarait un tableau, il fallait donner sa taille (plus exactement son nombre d'éléments). Mais cela nécessite de connaître ce paramètre *avant* d'écrire le programme. Or ceci n'est pas toujours possible (par exemple, si l'on fait un programme qui doit calculer la moyenne d'une classe, on ne connaît pas à l'avance le nombre d'élève, qui peut varier d'un classe à l'autre). Une solution consiste à réserver une place supérieure au maximum que l'on pourra renconter. Si cela est possible dans le cas précédent (une classe aura toujours moins de 100 élèves), ce n'est pas toujours le cas. Par exemple, si l'on fait un programme qui gère une bibliothèque, de nouveaux ouvrages arrivent tous les jours, et aucun maximum n'est envisageable. Ou alors le maximum que l'on peut considerer fera que l'on occupera toute la memoire de l'ordinateur.

Aussi, il apparaît plus judicieux de ne réserver la place qu'une fois le besoin connu (à l'exécution). Ceci est possible grâce à la fonction *calloc*, dont l'utilisation nécessite l'inclusion du fichier *stdlib.h*[15]. Ces fonctions essaient de réserver en mémoire la place demandée. Si c'est possible, elles renvoient un pointeur sur l'emplacement réservé. Sinon, elles renvoient le pointeur NULL (zéro).

Exemple :

```
#include<stdio.h>
#inclued<malloc.h>

main()
{
        (float *) notes; /* tableau des notes */
        int n = 0; /* nombre d'eleves */
        int i; /* compteur */
        float s = 0; /* somme des notes deja rentrees */

        printf("Nombre de notes a rentrer :");
        scanf("%d", &n);

        if (notes = calloc(n, size_of(float)))
```

15. sous Unix, c'est en fait en général malloc.h qu'il faut inclure

```
        {
                for(i = 0 ; i < n ; i++)
                {
                        printf("\nnote de l'eleve %d : ", i);
                        scanf("%f", &notes[i]);
                        s += notes[i];
                }
                printf("Moyenne des eleves : %e\n", s/n);
        }
        else
                printf("Calcul impossible : pas assez de memoire\n");
}
```

On remarquera que dans le cas présent, le tableau de notes n'est pas nécessaire. Cependant, si l'on désire archiver les notes, faire des moyennes par élèves, etc., un tel tableau devient très vite nécessaire.

1.14.2 Bien gérer l'allocation dynamique

Une allocation à la demande comme montrée précédemment est appelée *allocation dynamique*. Trois fonctions permettent une allocation dynamique :

- malloc : fonction qui prend en argument la taille de l'objet pour lequel il faut réserver de la place. Cette place peut être connue grâce à la fonction *size_of*. Cette fonction s'utilise en général pour les structures ;

- calloc(n, t) : permet de réserver de la place pour n objets de taille t . S'utilise en général pour les tableaux ;

- realloc(p, t) : re-attribue de la memoire à l'objet pointé par p. Si la nouvelle taille est plus petite que l'ancienne, la fin est coupée, le début est conservé. Si cette taille est plus grande, le début est conservé, la fin n'est pas initialisée.

Dans tous les cas, s'il y a échec, c'est la valeur nulle (0, ou NULL) qui est retournée par la fonction.

Toutes ces fonctions permettent de réserver de la mémoire. Mais si on peut ainsi *prendre* de la mémoire, ... il faut aussi pouvoir la rendre. C'est le rôle de la fonction *free(p)*, qui libère la mémoire pointée par p [16].

16. Si p est le pointeur NULL, free(p) ne fait rien

Note : La gestion dynamique de la mémoire est un point crucial de la programmation en C. En effet, les appels aux fonctions d'allocation mémoire sont relativement lents, et ne doivent donc pas être effectués sans raison. De plus, garder de la mémoire réservée alors qu'on ne s'en sert plus est inutile, et peut gêner d'autres programmes. Très souvent, on doit donc établir un compromis entre place mémoire et vitesse.

Note : Il ne faut jamais utiliser un objet dont la place en mémoire n'a pas été réservée, ou a été libérée. De plus, un programme propre devra, à la fin de son exécution, avoir libéré toute la place qu'il a eu l'occasion de réserver. Par exemple, dans le programme précédent, il faudrait rajouter, à la fin, l'instruction free(Notes).

1.14.3 Allocation dynamique et types récursifs

On a vu que dans le cas des types récursifs, il fallait aussi avoir déclaré les variables sur lesquelles pointeraient les pointeurs rendant la structure récursive. Mais ceci avait pour unique rôle de réserver la place en mémoire. Car l'intérêt de cette méthode est limité : on serait obligé de déclarer 1000 variables (ou un tableau de mille éléments). Avec l'allocation dynamique de mémoire, on n'a plus besoin de procéder ainsi...

Exemple :

```
#include<stdio.h>
#include<malloc.h>

struct pers
{
        char nom[20];
        char prenom[20];
        (struct pers *) pere;
        (struct pers *) mere;
}

(struct pers *) creer(void)
{

(struct pers *) nouveau;

        if (nouveau = malloc(size_of(struct pers)))
        {
                printf("nom de la personne : ");
                scanf("          printf("prenom de la personne : ");
                scanf("          nouveau->pere = NULL;
```

```
                nouveau->mere = NULL;
        }
        return nouveau;
}

main()
{
        int sortir = 0;
        char c;
        int correct;
        (struct pers *) courant;
        struct pers moi;
        strcpy(moi.nom, "MonNom");
        strcpy(moi.prenom, "MonPrenom");
        moi.pere = NULL;
        moi.mere = NULL;
        courant = &moi;

        while (!sortir)
        {
                printf("Nous traitons %s %s\n", courant->prenom, courant->nom);
                if (!courant->pere)
                        printf("pere inconnu\n");
                if (!courant->mere)
                        printf("mere inconnue\n");
                printf("Desirez-vous :\n 0. sortir\n1. Revenir a la source\n")
                printf("2. passer au pere“n3. passer a la mere\n")
                        correct = 0;
                while (!correct)
                {
                        c = getchar();
                        correct =
                                (c == '0')
                        ||      (c == '1' && courant != moi)
                        ||      (c == '2')
                        ||      (c == '3');
                }
                switch (c)
                {
                        case 0:
                                sortir = 1;
                                break;
                        case 1:
                                courant = moi;
                                break;
                        case 2:
```

```
                        if (!courant->pere)
                                courant->pere = creer();
                        courant = courant->pere;
                        break;
                case 3:
                        if (!courant->mere)
                                courant->mere = creer();
                        courant = courant->mere;
                        break;
                default:
                        break;
        }
        if (courant == NULL)
        {
                printf("Attention : plus assez de memoire;");
                printf("On revient a la racine\n.");
                courant = moi;
        }
    }
}
```

1.15 Lire et écrire dans des fichiers

La gestion des fichiers étant une gestion d'entrées-sorties, il faut inclure le fichier *stdio.h*. Les accès au fichier se font par l'intermédiaire d'un pointeur sur fichier (type FILE *).

Un fichier, avant d'être utilisé, doit être ouvert par l'intermédiaire de la fonction *fopen*, qui prend en paramètre le nom du fichier et les opérations autorisées dessus, et qui retourne un pointeur sur fichier. Après utilisation, un fichier doit être fermé par *fclose*.

Les principales fonctions d'écriture dans un fichier sont :

- *fprintf*, analogue à printf, mais le premier paramètre est un pointeur sur fichier ;
- *fputc*, analogue à putchar, mais deuxième paramètre est un pointeur sur fichier ;
- *fputs* écrit la chaîne passée en premier paramètre dans le fichier passé en second paramètre.

Les principales fonctions de lecture dans un fichier sont :

- *fscanf*, analogue à scanf, mais le premier paramètre est un pointeur sur fichier ;

- *fgetc*, analogue à getchar, mais deuxième paramètre est un pointeur sur fichier ;
- *fgets* lit une ligne dans le fichier passé en second paramètre et la stocke dans le tableau de caractères passé en premier paramètre.

Les différents mode d'ouverture d'un fichier sont les suivants :

- "r" lecture simple ;
- "w" création et écriture. Écrase une éventuelle ancienne version ;
- "a" Se place en écriture à la fin d'un fichier. Si le fichier n'existait pas, le crée ;
- "r+" lecture et écriture ;
- "w+" création, puis ouverture en lecture et écriture ;
- "a" se place en fin de fichier en mode lecture et écriture.

Note : les données écrites dans un fichier ne sont pas forcément recopiées immédiatement sur le disque. Pour forcer cette écriture avant le fclose, on peut utiliser la fonction fflush.

Exemple 1 : écriture dans un fichier

```
#include<stdio.h>

main()
{
        FILE *fp;
        int i;

        fp = fopen("Monfichier", "w");

        if (fp)
        {
                for(i = 1; i <= 100; i++)
                        fprintf(fp, "%4d%c", i, (i%10)? '\t': '\n');
                fclose(fp);
        }
}
```

Exemple 2 : lecture d'un fichier

```
#include<stdio.h>

main()
{
        FILE *fp;
        char chaine[100];

        fp = fopen("Monfichier", "r");

        if (fp)
        {
                while(!(fscanf(fp, "%s", &chaine) == EOF))
                        printf("%s\n", chaine)
                fclose(fp);
        }
}
```

1.16 Fonctions en paramètre

1.16.1 Pourquoi des fonctions en paramètre

Lorsque l'on crée certaines structures, en C, ainsi que les fonctions qui s'y rattachent, on aimerait pouvoir réutiliser ces fonctions pour des données d'un type différent. Par exemple, transformer une liste chaînée de chaîne de caractères en liste chaînée d'une structure comportant une chaîne et un entier. Le problème est que certaines fonctions (telles celles de comparaison) sont propres au type de donnée. Ces fonctions doivent donc être des paramètres. Le passage d'une fonction en paramètre ce fait par un passage de pointeur sur fonction.

1.16.2 Application

Il faut bien évidemment définir le type de fonction qui est attendu dans la liste des paramètres de la fonction qui "recevra" une fonction en paramètre.

En ce qui concerne le passage d'une fonction, il suffit de passer son nom : c'est déjà un pointeur sur fonction.

Exemple :

```
#include<stdio.h>

int fonc_en_param(int a, int b)
```

```
{
        return (a+b);
}

int recoit_fonc( int (*f)(int, int) , int a, int b)
{
        return f(a,b);
}

main()
{
        int a=3;
        int b=5;
        int c;

        c = recoit_fonc(fonc_en_param, a, b);


printf("%d\n", c);
}
```

1.17 Les définitions : *#define*

L'instruction *#define* permet d'effectuer une substitution dans le corps du fichier source, préalablement à son analyse.

L'utilisation de base est la déclaration de constantes. Par exemple :

```
#define TAILLE_ABS 10
#define TAILLE_ORD 20
#define TAILLE_GLOBALE (TAILLE_ABS * TAILLE_ORD)

int tableau[TAILLE_ABS][TAILLE_ORD];
```

On peut aussi paramétrer les macros par des arguments. Par exemple :

```
#define pair(x) ((x) % 2? 0 : 1)
```

Note : Attention de toujours parenthéser le texte d'une définition ainsi que les occurences des paramètres dans le corps de la définition. Sinon, on peut avoir des surprises... Exemple : si l'on définit

$$\#define\ carre(x)\ x * x$$

alors $carre(x+1)$ sera transformer en $x + 1 * x + 1$, ce qui, étant donné la priorité des opérateurs, vaut $x + (1 * x) + 1$

1.18 La programmation modulaire en C

1.18.1 Pourquoi la programmation modulaire

Qui dit "programmation modulaire" dit "programmation en modules". En C, cela correspond à diviser les sources d'un programmes en plusieurs fichiers. Cela offre plusieurs avantages :

- réutilisabilité ;
- facilité de compréhension ;
- travail en équipe;
- compilation fractionnée ;
- ...

Toutefois, lorsqu'une fonction d'un fichier fait appel à une fonction d'un autre fichier, il faut, à la compilation, connaître certaines choses sur la fonction appelée, notamment :

- être sûr qu'elle existe ;
- connaître son profil.

1.18.2 Rôle des fichiers d'en-tête

Ces fichiers servent à préciser le profil des fonctions d'un fichier source accessibles depuis d'autres. Il consistera principalement en la déclaration d'en-tête de fonctions. On trouvera aussi dans ce fichier la définition de certaines structures de données. Il constitue la *partie publique* de ce qu'il définit, la partie privée étant dans le fichier en ".c" associé. Par conséquent, le fichier ".h" doit aussi contenir tous les commentaires nécessaires pour comprendre à quoi ça sert, et comment utiliser les fonctions qui y sont définies.

Toutefois, on ne peut pas définir dans deux sources d'un même projet une même fonction. Or on peut avoir besoin, dans deux sources différents, de faire appel aux fonctions d'un même troisième fichier source. Pour éviter alors ce

problème de multiple déclaration des fonctions, un fichier d'en-tête fait en général usage des directives `#ifndef` et `#endif`.

Exemple : en-tête d'un fichier implantant des fonctions sur les listes (sans commentaires pour gagner de la place) :

```
/* Bibliotheque de fonctions sur les listes */
#ifndef BIBLIOLISTE_H
#define BIBLIOLISTE_H
typedef struct ll1
        {
        element val;
        suivant *ll1;
        } liste1;

typedef struct ll2
        {
        liste1 *premier;
        } liste2;

liste2 *creer(void)
liste2 *adjq(element e1, liste2 *ancienne_liste)

destruction (liste2 *ll)

liste2 *adjt(element el, liste2 *ancienne_liste)
liste2 *conc(liste2 *l1, *l2)

int appartient(element el, liste2 *l)
int nb_occ(element el, liste2 *l)

liste2 *suppq(liste2 *ancienne_liste)
liste2 *suppr_1(element el, liste2 *l)
liste2 *suppr(element el, liste2 *l)
#endif
```

1.18.3 Variables globales

Un fichier C peut avoir besoin de faire référence à des variables globales déclarées dans un autre fichier. Auquel cas, dans le fichier "utilisateur", il faut déclarer la variable global, mais en précisant qu'elle est *externe*.

exemple :

```
extern int toto;
```

1.19 Fonctions liées aux chaînes de caractères

1.19.1 Introduction

Comme on l'a vu, il n'existe pas en C à proprement parler de type *chaîne de caractères*. Ceci fait que le langage n'a pas d'opérateurs où mots-clefs réservés pour le traitement de celles-ci. Cependant ce manque est en partie comblé par des fonctions de la libraire de base, que l'on peut utiliser en incluant le fichier `string.h`.

1.19.2 Quelques fonctions

On se limitera ici aux fonctions les plus utiles :

- `char *strcpy(char *dest, char *src)` : Copie la chaîne *src* dans la chaîne *dest*, y compris le caractère nul de fin de chaîne. De plus, cette fonction retourne *dest*. L'allocation pour *dest* doit avoir été faite avant.
- `char *strdup(char *src)` : même rôle que `strcpy`, mais fait l'allocation mémoire.
- `char *strncpy(char *dest, char *src, size_t n)` : comme `strcpy`, mais copie au plus n caractères. Complète par des caractères nuls si la chaîne n'est pas assez longue.
- `size_t strlen(char *src)` : renvoie la longueur de la chaîne *src*.
- `char *strcat(char *dest, char *src)` : rajoute la chaîne *src* à la chaîne *dest*, et retourne cette dernière. N.B. : il existe aussi `strncat`.
- `int strcmp(char *ch1, char *ch2)` : compare les chaînes *ch1* et *ch2* en utilisant l'ordre alphabétique. Renvoie un nombre négatif si $ch1 < ch2$, un nombre positif si $ch1 > ch2$, et zéro si $ch1 = ch2$. N.B. : il existe aussi `strncmp`.
- `char *strcstr(char *ch, char *ss_ch)` : retourne un pointeur sur la première occurence de *ss_ch* dans *ch*. Si elle n'est pas présente, renvoie le pointeur `NULL`.

1.20 Les arguments de la ligne de commande

1.20.1 Introduction

Dans de nombreux systèmes, les programmes sont appelés par l'intermédiaire d'une *ligne de commande*, sur laquelle on tape le nom du programme, auquel on peut associer des paramètres. Exemple : lorsque l'on tape `cp toto1 toto2`, `toto1` et `toto2` sont des paramètres du programme `cp`. Il faut donc que le programme puisse récupérer ses paramètres. Le langage C ayant été conçu notamment dans le but d'écrire Unix et les programmes qui tournent dessus, il a donc été prévu de pouvoir récupérer ces paramètres.

1.20.2 Les paramètres `argc` et `argv`

La fonction que l'ont déclare généralement sous la forme `main()`, a en fait le profil suivant : `int main (int argc, char *argv[])`, où `argc` est un entier qui indique le nombre de paramètres de la ligne de commande (nom du programme y compris), et `argv` est un tableau de chaîne de caractères, une par argument. À noter cependant que le premier élément du tableau `argv`, d'indice zéro, est le nom de la commande.

Exemple :

```
main(int argc, char *argv[])
{
        int i;

        printf("Vous avez lance le programme: %s ",argv[0]);
        switch (argc)
                {
                case 1 :
                        printf("sans argument\n");
                        break;
                case 2 :
                        printf("avec l'argument %s\n", argv[1]);
                        break;
                default :
                        printf("avec les arguments suivants\n");
                        for ( i = 1; i <= argc ; i++)
                                printf("%s\n", argv[i]);
                }
}
```

1.21 Gestion des processus

1.21.1 Introduction

Un système tel qu'Unix est un système qui permet d'avoir plusieurs processus en même temps. On donne ainsi à l'utilisateur l'impression d'avoir la possibilité de faire plusieurs choses à la fois. En fait, au niveau machine, si celle-ci n'a qu'un seul processeur, il n'en est rien. Mais on peut ainsi notamment utiliser les temps morts de certaines applications pour les autres. C ayant été créé pour Unix, il doit donc permettre de gérer les processus.

1.21.2 Création d'un processus

La création d'un processus en C se fait à partir de la fonction `fork`. Celle-ci renvoie -1 si elle échoue. Si elle réussit, elle crée un processus *fils* du premier. Celui-ci dispose d'un environnement semblable à celui de son père. Le fils poursuit l'exécution du programme après le `fork()` tout comme son père. La seule différence est dans le résultat de la commande `fork()`. Pour le père, on trouve le numéro de processus affecté au fils. Pour le fils, on obtient 0.

Exemple :

```
main()
{
        int t,i;

        t = fork();
        if (t == -1)
                {
                printf("Erreur : creation fils a echouee\n");
                return 1;
                }
        if (t)
                {
                printf("On est dans le processus pere\n");
                printf("Et on vient de creer un fils numero %d\", t);
                for (i = 0; i <= 200 ; printf("%4d",i++));
                }
        else
                {
                printf("On est dans le processus fils\n");
                printf("cree par un pere numero %d\n",getppid());
                for (i = 400; i <=600 ; printf("%d",i++));
                }
}
```

Dans l'exemple précédent, si le père se termine avant le fils, celui-ci est adopté par le processus *init*, d'identificateur 1. Pour éviter cela, Il faut que le processus père attende la mort de son fils. Cela se fait grâce à la fonction `wait` par un :

$$r = wait(\&id);$$

où id est l'identificateur du processus dont ont attend la mort. r est tel que :

- r / 256 = paramètre de exit ;
- r mod 256 = numero signal si processus tué par un signal sans création d'un fichier core ;
- r mod 256 = 128 + n^o du signal si processus tué par un signal avec création d'un fichier core.

Dans le cas où l'on souhaite qu'un processus lance un autre programme, mais sans que cela soit un autre processus, il faut utiliser les fonction `execl` ou `execv`, ou une de leur variante. `execl` suppose que le nombre de paramètre est connu : on passe le nom du programme en paramètre, puis tous les paramètres du programme sous forme de chaînes de caractères. Dans le cas d'`execv`, on passe un tableau de chaînes de caractères. Dans les deux cas, le premier paramètre est toujours le nom du programme.

Exemple : lancement du programme `prog2` sans paramètre.

```
execl("prog2", "prog2");
```

Les fonctions `execle` et `execve` jouent le même rôle que `execl` et `execv`, mais un dernier paramètre est un tableau `envp` qui permet de passer un environnement.

Enfin, les fonctions `execlp` et `execvp` sont similaires à `execl` et `execv`, mise à part le fait qu'elles permettent d'exécuter un programme situé dans un répertoire quelconque de la variable d'environnement PATH.

1.21.3 Communication entre processus

Si l'on désire avoir plusieurs processus qui collaborent à une tâche donnée, il faut qu'ils puissent communiquer. Il existe plusieurs moyens pour faire communiquer des processus. Ici, seule la communication par *pipes* (tubes) est présentée.

Un pipe est une implantation d'une file. Il a une taille limitée. Un processus qui essaie d'écrire dans un pipe plein est bloqué jusqu'à ce qu'il y ait une place de libérée.

La *production* et la *consommation* au niveau d'un pipe se font par l'intermédiaire de descripteurs de fichier (un par opération), que le processus doit connaître. Un tube peut bien évidemment être fermé par un processus. Dans la mesure où plusieurs processus peuvent partager un même tube à la fois en lecture et en écriture, un tube ne transmettra un EOF (End Of File) que lorsque tous les processus auront fermé le tube en écriture.

La création d'un pipe se fait au moyen de la commande `n = pipe(p)`, où p est un tableau de 2 entiers, et n vaut 0 si l'opération s'est passée sans problème.

L'écriture dans un tube utilise la commande `write(p[1], &c, n)` où &c est un pointeur sur caractère, et n est le nombre de caractère à transmettre.

La lecture dans un tube se fait par la commande `read(p[0], &c, n)`. Si le tube est vide, elle renvoie zéro.

Un processus peut fermer les accès en lecture ou écriture dans un tube au moyen respectivement des commandes `close(p[0])` et `close(p[1])`.

Plutôt que d'utiliser les fonctions `read` et `write` pour accéder à un pipe, on peut utiliser les fonctions classiques sur les fichiers telles que `fprintf` et `fscanf`. Cela nécessite cependant d'obtenir les pointeurs sur fichier nécessaires au moyen de la fonction `fdopen` :

```
lecture = fdopen(p[0], "w");
ecriture = fdopen(p[1], "r");

...

fprintf(lecture, "%d ",n);

...

fscanf(ecriture, "%d", &m);

...

fclose(lecture);
fclose(ecriture);
```

Chapitre 2

Un peu de théorie

2.1 Preuve de programme

2.1.1 Introduction

Lorsqu'on écrit un programme, il peut être intéressant (!) de s'assurer qu'il fait bien ce que l'on veut. Or ceci peut être plus ou moins évident...

Exemple :

```
#include <stdio.h>

main()
{
        int n=0;
        int rac=0;

        printf("Entrez un nombre: ");
        scanf("%d", &n);
        rac = n;
        while (rac * rac > n)
                rac--;
        printf("\nrac = %d\n", rac);
}
```

Deux questions se posent pour le programme précédent :

- ce programme, lorsqu'il se termine, calcule-t-il bien la racine carrée du nombre saisi?
- ce programme se termine-t-il toujours?

À ces deux questions sont liées les notions de correction partielle et de correction totale. On dit qu'un programme est *partiellement correct* si, lorsqu'il se termine, le résultat obtenu est le résultat désiré. Un programme est *totalement correct* s'il se termine toujours et qu'il produit le résultat désiré.

2.1.2 Méthode de Floyd

Introduction

Cette méthode est l'une des méthodes qui permet de s'assurer de la correction d'un programme. Elle consiste à étiqueter les différents points de contrôle du programme, et à y mettre des assertions.

Un programme est correct (partiellement) si la première assertion est une conséquence de Ψ, pré-condition du problème, si toute assertion se déduit de la "précédente", et si la dernière implique Φ, post-condition du problème.

Un programme est exempt d'erreur à l'exécution si chaque assertion établit que les variables appartiennent bien au domaine de l'instruction qui suit.

Un programme est totalement correct si le programme est partiellement correct, qu'il n'y a pas d'erreur à l'exécution et qu'il termine.

Exemple

La difficulté de la preuve d'un programme dépend beaucoup de la manière dont on l'écrit, tout comme sa compréhension. Par exemple, le programme suivant :

```
#include <stdio.h>
#define MAX 1000
main()
{
        int Tableau[MAX];
        int k, marqueur;

        for(k=0;k<MAX;Tableau[k++]=1);
        for(k=2;k<=MAX/2;k++)
                if (Tableau[marqueur=k])
                        while((marqueur+=k)<MAX)
                                Tableau[marqueur]=0;
        for(k=2;k<MAX;k++)
                if(Tableau[k])
                        printf("%d\t",k);
        printf("\n");
}
```

est plus facile à comprendre une fois réécrit sous la forme suivante :

```
/* Programme recherchant les nombres premiers de 1 a n */
/* par la methode du crible d'Eratosthene */
#include <stdio.h>
#define MAX 1000

main()
{
        int Tableau[MAX];
        int k, marqueur;
        /* On initialise le tableau a 1 : tout nombre est considere */
        /* premier tant qu'il n'a pas ete trouve comme multiple d'un autre */
```

```
        for(k=0;k<MAX;k++])
                Tableau[k] = 1;

        /* Le crible proprement dit : on commence par 2 */
        /* On s'arrete a MAX/2 (Les nombres au-dessus non rayes */
        /* n'ayant pas de multiple inferieur a MAX) */
        for(k=2;k<=MAX/2;k++)
                /* On teste si le nombre n'a pas ete marque (il est*/
                /* donc premier). Si c'est le cas, on va rayer ses multiples */
                /* Sinon, il n'y a rien a faire (ses multiples ont deja ete rayes) */
                if (Tableau[k])
                {
                        /* On initialise le marqueur au premier multiple de l'element courant */
                        marqueur = 2*k;
                        /* On raye tous les multiples: */
                        while(marqueur<MAX)
                        {
                                Tableau[marqueur]=0;
                                /* marqueur pointe sur le multiple suivant */
                                marqeur +=k;
                        }
                }
        /* On a fini : on va afficher les resultats */
        for(k=2;k<MAX;k++)
                /* Le nombre est-il premier? */
                if(Tableau[k])
                        /* Oui, alors on l'affiche */
                        printf("%d\t",k);
        printf("\n");
}
```

Pour faire la preuve de notre programme, on commence par définir ce que l'on attend de lui :

- précondition d'exécution :

$$\Psi == true$$

- postcondition :

$$\Phi == \begin{cases} Tableau[2] = 1 \\ \forall i \in 3..MAX-1, Tableau[i] = 0 \Leftrightarrow \exists j \in 2..i-1 / j \text{ divise } i \end{cases}$$

Maintenant, on étiquette le programme :

```
/* Programme recherchant les nombres premiers de 1 a n */
```

```
/* par la methode du crible d'Eratosthene */
#include <stdio.h>
#define MAX 1000

main()
{
        int Tableau[MAX];
        int k, marqueur;

        /* On initialise le tableau a 1 : tout nombre est considere */
        /* premier tant qu'il n'a pas ete trouve comme multiple d'un autre */
        for(k=0;k<MAX;k++])

                ⓪∀j ∈ 0..k − 1, Tableau[j] = 1

                Tableau[k] = 1;

        ①∀j ∈ 3..MAX − 1, Tableau[j] = 1

        /* Le crible proprement dit : on commence par 2 */
        /* On s'arrete a MAX/2 (Les nombres au-dessus non rayes */
        /* n'ayant pas de multiple inferieur a MAX) */
        for(k=2;k<=MAX/2;k++)

                ②∀j ∈ 3..MAX − 1, Tableau[j] = 0 ⇔ ∃i ∈ 2..k − 1/i divise j

                /* On teste si le nombre n'a pas ete marque (il est*/
                /* donc premier). Si c'est le cas, on va rayer ses multiples */
                /* Sinon, il n'y a rien a faire (ses multiples ont deja ete rayes) */
                if (Tableau[k])
                {
                        /* On initialise le marqueur au premier multiple de l'element courant */
                        marqueur = 2*k;
                        /* On raye tous les multiples: */
                        while(marqueur<MAX)
                        {

                                ③∀j ∈ 3..MAX − 1, Tableau[j] = 0 ⇔
                                (∃i ∈ 2..k − 1/i divise j ∨ (j < marqueur ∧ k divise j ∧ j > k))

                                Tableau[marqueur]=0;
                                /* marqueur pointe sur le multiple suivant */
                                marqeur +=k;
                        }
                }
```

$$④ \forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \exists i \in 2..j-1 / i \text{ divise } j$$

```
        /* On a fini : on va afficher les resultats */
        for(k=2;k<MAX;k++)
                /* Le nombre est-il premier? */
                if(Tableau[k])
                        /* Oui, alors on l'affiche */
                        printf("%d\t",k);
        printf("\n");
}
```

Il ne reste plus qu'à faire les preuves... On remarquera d'ailleurs que les assertions données ici ne sont pas suffisantes.

Quelques règles de preuve

- affectation :

$$\begin{array}{l} \Phi_i \\ \qquad var = exp \\ \Phi_{i+1} \end{array}$$

  Règle associée :

$$Phi_i \Rightarrow [exp/var]Phi_{i+1}$$

- test *if* :

$$\begin{array}{l} \Phi_i \\ \qquad if(cond) \\ \qquad\qquad \Phi\_i_1 \\ \qquad\qquad inst_1 \\ \qquad\qquad \Phi\_i_2 \\ \qquad else \\ \qquad\qquad \Phi\_i_3 \\ \qquad\qquad inst_2 \\ \qquad\qquad \Phi\_i_4 \\ \Phi_{i+1} \end{array}$$

  Règles associées :

$$\begin{array}{l} \Phi_i \wedge cond \Rightarrow \Phi_{i_1} \\ \Phi_i \wedge \neg cond \Rightarrow \Phi_{i_3} \\ \Phi_{i_2} \Rightarrow \Phi_{i+1} \\ \Phi_{i_4} \Rightarrow \Phi_{i+1} \end{array}$$

- Boucle *while* :

Φ_i
$\quad while(cond)$
$\quad\quad \Phi_{i_1}$
$\quad\quad inst$
$\quad\quad \Phi_{i_2}$
Φ_{i+1}

Règles associées :

$$\Phi_i \wedge cond \Rightarrow \Phi_{i_1}$$
$$\Phi_i \wedge \neg cond \Rightarrow \Phi_{i+1}$$
$$\Phi_{i_2} \wedge cond \Rightarrow \Phi_{i_1}$$
$$\Phi_{i_2} \wedge \neg cond \Rightarrow \Phi_{i+1}$$

Attention : ces règles (non exhaustives) ne sont valables que dans la mesure où l'on ne pratique aucune affectation dans les phases de test !

Preuve de correction partielle: Application

On va appliquer la méthode à la boucle *while* du programme précédent.

- Première partie : Initialisation
  - Hypothèses

$$\forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \exists i \in 2..k-1 \text{ t.q. } i \text{ divise } j$$
$$Tableau[k] = 1$$
$$marqueur = 2 * k$$
$$marqueur < MAX$$

  - Conclusion

$$\begin{cases} marqueur > k \\ k \text{ divise } marqueur \\ \forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \begin{pmatrix} \exists i \in 2..k-1 \text{ t.q. } i \text{ divise } j \vee \\ j < marqueur \wedge k \text{ divise } j \wedge j > k \end{pmatrix} \end{cases}$$

- Deuxième partie : Bouclage
  - Hypothèses

$$\begin{cases} marqueur + k < MAX \\ marqueur > k \\ k \text{ divise } marqueur \\ \forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \begin{pmatrix} j = marqueur \vee \\ \exists i \in 2..k-1 \text{ t.q. } i \text{ divise } j \vee \\ j < marqueur \wedge k \text{ divise } j \wedge j > k \end{pmatrix} \\ marqueur' = marqueur + k \\ Tableau' = Tableau \lessdot (Tableau[k] = 0) \end{cases}$$

- Conclusion

$$\begin{cases} marqueur' > k \\ k \text{ divise } marqueur' \\ \forall j \in 3..MAX-1,\ Tableau'[j] = 0 \Leftrightarrow \begin{pmatrix} \exists i \in 2..k-1 \text{ t.q. } i \text{ divise } j \vee \\ j < marqueur' \wedge k \text{ divise } j \wedge j > k \end{pmatrix} \end{cases}$$

- Sortie sans rentrer dans la boucle :

  - Hypothèses

$$\begin{array}{l} \forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \exists i \in 2..k-1 \text{ t.q. } i \text{ divise } j \\ Tableau[k] = 1 \\ marqueur = 2 * k \\ marqueur >= MAX \end{array}$$

  - Conclusion

$$\forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \exists i \in 2..k/i \text{ divise } j$$

- Sortie en étant rentré dans la boucle :

  - Hypothèses

$$\begin{cases} marqueur' \geq MAX \\ marqueur' > k \\ k \text{ divise } marqueur' \\ \forall j \in 3..MAX-1,\ Tableau'[j] = 0 \Leftrightarrow \begin{pmatrix} \exists i \in 2..k-1 \text{ t.q. } i \text{ divise } j \vee \\ j < marqueur' \wedge k \text{ divise } j \wedge j > k \end{pmatrix} \end{cases}$$

  - Conclusion

$$\forall j \in 3..MAX-1, Tableau[j] = 0 \Leftrightarrow \exists i \in 2..k/i \text{ divise } j$$

2.1.3 Preuve de terminaison

Structure bien fondée

Une structure bien fondée $(\mathcal{D}, <)$ est telle que :

- $<$ est une relation d'ordre sur $\mathcal{D}$;
- il n'existe pas de suite infinie décroissante sur $\mathcal{D}$ muni de $<$.

Un exemple classique de structure bien fondée est $(\mathbb{N}, <)$.

Application à la terminaison des boucles

Pour prouver qu'un programme termine, il faut et il suffit de prouver que toutes les boucles terminent. Pour cela, on définit une fonction f sur l'ensemble des variables du programme et à valeur dans $\mathcal{D}$. Il suffit alors de vérifier qu'à chaque itération, l'exécution du corps de la boucle fait décroître la valeur de f appliquée aux variables du programme par rapport à une structure bien fondée $(\mathcal{D}, <)$.

Preuve de terminaison: Application

Dans le programme précédent, on cherche à montrer la terminaison de la boucle *while*.

On choisit comme structure bien fondée la structure $(\mathbb{N}, <)$. La fonction choisie est la fonction f définie ainsi : $f(var) = max(0, (MAX - marqueur))$.

- f toujours définie, à valeur dans $\mathbb{N}$

$$marqueur < MAX \Rightarrow max(0, MAX - marqueur) \in \mathbb{N}$$

- f décroit :

$$k \in \mathbb{N}^* \wedge marqueur < MAX \Rightarrow$$
$$max(0, MAX - marqueur - k) < max(0, MAX - marqueur)$$

2.2 Notion de complexité des algorithmes

2.2.1 Introduction

Lorsque des programmes doivent être exécutés sur de nombreuses données, le temps d'exécution devient un critère vital. Celui-ci dépend de différents paramètres, comme la puissance du microprocesseur de la machine, la vitesse de lecture/écriture du disque dur, la taille données, etc. ce dernier critère peut, contrairement aux autres, être étudié sur le programme (ou l'algorithme) lui-même, indépendamment de l'implantation.

On essaie alors d'évaluer le temps d'exécution par exemple en nombre d'instructions exécutées. Ce temps sera en général exprimé sous la forme d'une fonction, qui dépendra notamment de n, nombre des données.

Le paramètre intéressant dans cette fonction est son *ordre*: il permet d'estimer comment va évoluer le temps de calcul avec la taille des données. Ceci permet de tester le programme à petite échelle, tout en ayant une idée du temps que cela mettra sur le problème en taille réelle.

C'est cet ordre que l'on appel *complexité de l'algorithme*.

2.2.2 Petit rappel : Ordre d'une fonction

Lorsque l'on cherche à evaluer l'ordre d'une fonction, on "supprime" tout ce qui devient négligeable à l'infini. On donne alors l'ordre sous la forme $O(f)$ (lire "grand 'o' de f), où f est une fonction "simple".

Exemple : la fonction $3n(\log(n) + 2n - 1)$ est en $O(3n\log(n))$

2.2.3 Différentes mesures de complexité

La complexité d'un algorithme est en général exprimée en fonction de la taille des données. On distingue cependant là encore plusieurs types. Les principaux sont :

- Complexité moyenne
- Complexité au pire

En effet, un programme peut contenir des instructions de test, fonction des valeurs des données (inconnues *a priori*), dont l'évaluation induira l'exécution de séquences d'instructions de complexité variable. Par conséquent, la "complexité à l'exécution" dépend des données elles-mêmes. On évalue donc en général ce que l'on appelle la *complexité moyenne*. Cependant, pour vérifier que dans certains cas particuliers, on n'atteint pas des temps rédibitoires, il peut arriver que l'on calcule aussi la *complexité maximale*.

2.2.4 Que mesurer

Chaque instruction a son propre temps d'exécution. Par exemple, l'affectation est quasi-instantanée, l'addition et la soustraction sont tres rapides, la

multiplication un peu moins, la division est relativement lente, etc. On ne peut donc pas considérer chaque instruction de la même façon. Donc là aussi, on aura tendance à ne pas compter ce qui est négligeable.

Par exemple, dans un problème qui fait beaucoup de calculs, on négligera les affectations. Par contre, dans un algorithme de tri, qui ne fait que des tests et affectations, cela serait ridicule !

On peut ne pas forcément faire un décompte instruction par instruction. Ainsi, on s'intéresse souvent à des blocs d'instructions toujours exécutées d'un bout à l'autre. Par exemple, dans des algorithmes donnés sous forme d'une boucle, on pourra évaluer le nombre de fois où le corps de la boucle est effectué.

2.2.5 Exemple : Le tri à bulle

Algorithme

Le principe de l'algorithme ci-dessous est expliqué dans la section consacrée au tri.

```
void tri_a_bulle(int *tableau, int taille)
{
        int max;
        int i;
        int trie = 0;
        int temp;

        for (max = taille; max > 1 && !trie ; max--)
                {
                trie = 1;
                for( i = 0 ; i < max-1 ; i++)
                        if (tableau[i+1] < tableau[i])
                                {
                                temp = tableau[i];
                                tableau[i] = tableau[i+1];
                                tableau[i+1] = temp;
                                trie = 0;
                                }
                }
}
```

Mesure de complexité

Dans le programme ci-dessous, on prendra comme mesure de complexité le nombre de fois où le corps de la boucle interne est exécuté, et on évalue la complexité dans le pire des cas.

Au pire, le tableau ne se révèle trié qu'à la dernière étape. Or à la première étape, la boucle interne est exécutée $n-1$ fois. À l'étape suivante, $n-2$ fois, ... Et finalement, 1 fois à la dernière étape.

D'où la complexité au pire :

$$f(n) = \sum_{i=1}^{n-1} i$$

soit

$$f(n) = \frac{n(n-1)}{2}$$

soit

$$f(n) \approx n^2/2$$

2.3 Langage régulier

2.3.1 Définitions

Soit V un ensemble (*a priori* fini). On appelle *mot* sur V tout suite finie d'éléments de V. On note V^* l'ensemble des mots sur V. On appelle langage sur V tout sous-ensemble de V^*.

Un mot α est une application de $[1, n]$ dans V, avec $n \in \mathbb{N}$. On appelle n *longueur* de α, et on note $n = |\alpha|$. Plutôt que d'utiliser la notation fonctionnelle, on notera α ainsi :

$$\alpha = a_1 a_2 \ldots a_n$$

On appelle *mot vide* le seul mot de V^* de longueur zéro, et on le note $\wedge$.

On note $\gamma = \alpha\beta$ la concaténation des mots $\alpha = a_1 a_2 \ldots a_m$ et $\beta = b_1 b_2 \ldots b_n$ sur V, ainsi définie :

$$\gamma = a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n$$

Soit α un mot sur V. On appelle *facteur gauche* (resp. *facteur droit*) de α tout mot β tel qu'il existe γ vérifiant $\alpha = \beta\gamma$ (resp. $\alpha = \gamma\beta$).

On appelle *facteur* de α tout mot β tel qu'il existe deux mots γ et γ' tels que $\alpha = \gamma\beta\gamma'$.

2.3.2 Automates finis

On appelle automate fini sur V un quadruplet $\mathcal{A} = (S, s_0, r, S')$ tel que :

- S est un ensemble fini appelé *ensemble des états* de $\mathcal{A}$;
- s_0 est un élément de S appelé *état initial* de $\mathcal{A}$;
- r est une relation de type $S \times V \leftrightarrow S$, appelée *loi de transition* de $\mathcal{A}$;
- S' est un sous-ensemble de S appelé *ensemble des états de satisfaction* de $\mathcal{A}$.

Dans le cas où r est une fonction dont le domaine ne contient pas $\wedge$, on parle d'*automate fini déterministe*. On note $s.\alpha$ l'application de r à s et α.

Pour un automate $\mathcal{A}$, On définit une application $\phi_{\mathcal{A}}$ par :

$$\begin{cases} \phi_{\mathcal{A}}(\wedge) = s_0 \\ \phi_{\mathcal{A}}(\alpha a) = \phi_{\mathcal{A}}(\alpha).a \end{cases}$$

On appelle *langage reconnu par* $\mathcal{A}$ le langage $L(\mathcal{A})$ sur V défini par :

$$L(\mathcal{A}) = \phi_{\mathcal{A}}^{-1}(S')$$

On dit qu'un langage L sur V est un *langage régulier* si et seulement si il existe un automate fini $\mathcal{A}$ tel que $L = L(\mathcal{A})$ (on dit alors que $\mathcal{A}$ *reconnaît* L).

Le langage vide est régulier. De plus, tout langage fini est régulier.

2.3.3 Notation

On peut définir les langages réguliers en faisant notamment appel aux opérateurs suivants :

- ab : a suivi de b ;
- a^* : a répété un nombre fini (positif ou nul) de fois ;
- a^+ : a répété un nombre fini (strictement positif) de fois ;
- a^n : a répété n fois ($n \in \mathbb{N}$) ;
- {a,b,c} : a ou b ou c.

Exemple : L = {a, b}*b{c}+

2.3.4 Déterminisation d'un automate

Soit un automate $\mathcal{A} = (S, s_0, r, S')$ que l'on se propose de déterminiser en un nouvel automate $\mathcal{B} = (T, t_0, f, T')$. On procède ainsi :

$$t_0 = s_0 \cup \{s \in S / s_0.\wedge = s\}$$

Puis pour chaque mot α de V, on calcule $t_0.\alpha$, ensemble des $s_i.\alpha$, avec $s_i \in t_0$.

On obtient alors des ensembles d'états de S. Chaque nouvel ensemble ainsi obtenu se voit alors associé un nouvel état t_i de T. On réitère alors l'opération jusqu'à avoir envisagé tous les t_i obtenus avec tous les mots de V.

On considère comme état terminal tout état de T qui "contient" un état de S'.

Exemple : pour le langage L précédent, on peut construite l'automate :

$$\mathcal{A} = (\{s_0, s_1, s_2\}, s_0, \{(s_0, a, s_0), (s_0, b, s_0), (s_0, b, s_1), (s_1, c, s_2), (s_2, c, s_2)\}, \{s_2\})$$

On procède alors ainsi :

$$\begin{array}{l} t_0 = \{s_0\} \\ t_0.a = \{s_0\} = t_0 \\ t_0.b = \{s_0, s_1\} = t_1 \\ t_1.a = \{s_0\} = t\_0 \\ t_1.b = \{s_0, s_1\} = t_1 \\ t_1.c = \{s_2\} = t_2 \\ t_2.c = \{s_2\} = t_2 \end{array}$$

D'où le nouvel automate : $\mathcal{B} = (T, t_0, f, T')$ avec :

$$\left\{ \begin{array}{l} T = \{t_0, t_1, t_2\} \\ f = \{(t_0, a, t_0), (t_0, b, t_1), (t_1, a, t_0), (t_1, b, t_1), (t_1, c, t_2), (t_2, c, t_2)\} \\ T' = \{t_2\} \end{array} \right.$$

2.4 Langages algébriques

2.4.1 introduction

Les langages algébriques, apportent, par rapport aux langages réguliers, la gestion des mots *bien parenthésées* (structures de bloc notamment).

Définition 1 : Une *grammaire algébrique* G est un quadruplet $(N, T, \rightarrow, X)$ tel que :

- N est un vocabulaire, appelé *vocabulaire non terminal* ;
- T est un vocabulaire, appelé *vocabulaire terminal* ;
- N et T sont disjoints ;
- X est un élément de N appelé *axiome* ;
- $\rightarrow$ est une relation de N vers V^* (avec $V = N \cup T$) telle que pour tout mot A de N, le cardinal des images de A par $\rightarrow$ est fini.

Définition 2 : Soient u et v deux mots sur V. On dit que u *se réécrit* en v (et on note $u \rightarrowtail v$) si et seulement si on peut écrire $u = a'Ua''$ et $v = a'Va''$ avec $U \rightarrow V$.

Définition 3 : On dit que u *se dérive en* v s'il existe une suite finie $u_1, \ldots, u_n$ de mots de V tels que $u_1 = u$, $u_n = v$ et $u_i \rightarrowtail u_{i+1}$. On note cette relation $u \overset{*}{\rightarrowtail} v$.

Remarque : il s'agit d'une relation réflexive et transitive.

Définition 4 : Un langage est algébrique s'il peut être engendré par une grammaire algébrique c'est-à-dire s'il est égal au vocabulaire terminale d'une grammaire algébrique.

Exemple :

N = {EXP, NOMBRE, CHIFFRE}

T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /}

X = EXP

EXP $\rightarrow$ EXP + EXP | EXP - EXP | EXP * EXP | EXP / EXP | NOMBRE

NOMBRE $\rightarrow$ CHIFFRE NOMBRE | CHIFFRE

CHIFFRE $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |8 | 9

Exercice : calcul des *premiers* et *suivants* :

premiers(CHIFFRE) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

premiers(NOMBRE) = premiers(CHIFFRE) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

premiers(EXP) = premiers(NOMBRE) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

suivants(EXP) = {+, -, *, /}

suivants(NOMBRE) = suivants(EXP) = {+, -, *, /}

suivants(CHIFFRE) = premiers(NOMBRE) $\cup$ suivants(NOMBRE) = {+, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Chapitre 3

Structures de données

3.1 Listes chaînées

3.1.1 Introduction

Lorsque l'on doit stocker un ensemble de données identiques, les tableaux se révèlent intéressants. Cependant, ils présentent quelques inconvénients majeurs :

- ils sont difficilement extensibles ;
- l'insertion n'est possible que par un décalage de tous les éléments suivants.

Les listes chaînées, qui implantent le mécanisme des listes linéaires, permettent de pallier ces inconvénients.

3.1.2 Les listes linéaires

Les listes sont des *sortes* dotées de fonctions de création et d'axiomes définissant certaines opération de base.

- créateurs : *nil* et *adjq* sont les deux créateurs de liste. le premier correspond à la liste vide, le deuxième à l'adjonction en queue.
  - $nil : \rightarrow liste$
  - $adjq : element \times liste \rightarrow liste$

  Ainsi, la liste [1 ; 2 ; 3] est représentée par :

  $$adjq(3, adjq(2, adjq(1, nil)))$$

- quelques opérateurs de base :
  - *suppq* : suppression en queue :

    $$\begin{aligned} &suppq : liste \rightarrow liste \\ &suppq(nil) = \perp \\ &suppq(adjq(e, l)) = l \end{aligned}$$

  - *adjt* : adjonction en tête :

    $$\begin{aligned} &adjt : element \times liste \rightarrow liste \\ &adjt(e, nil) = adjq(e, nil) \\ &adjt(e_1, adjq(e_2, l)) = adjq(e_2, adjt(e_1, l)) \end{aligned}$$

  - *conc* : concaténation :

    $$\begin{aligned} &conc : liste \times liste \rightarrow liste \\ &conc(l, nil) = l \\ &conc(l_1, adjq(e, l_2)) = adjq(e, conc(l_1, l_2)) \end{aligned}$$

- *appartient* : appartenance d'un element à une liste :

$$appartient : element \times liste \rightarrow bool$$
$$appartient(e, nil) = false$$
$$appartient(e_1, adjq(e_2, l)) = ou(e_1 = e_2, appartient(e_1, l))$$

- *nb_occ* : nombre d'occurences d'un élément dans une liste :

$$nb\_occ : element \times liste \rightarrow nat$$
$$nb\_occ(e, nil) = 0$$
$$nb\_occ(e_1, adjq(e_2, l)) = plus(si(e1 = e2, 1, 0), appartient(e_1, l))$$

- *suppr_1* : suppression de la dernière occurence d'un élément :

$$suppr\_1 : element \times liste \rightarrow liste$$
$$suppr\_1(e, nil) = nil$$
$$suppr\_1(e_1, adjq(e_2, l)) = si(e_1 = e_2, l, adjq(e_2, suppr\_1(e_1, l)))$$

- *suppr* : suppression de toutes les occurences d'un élément :

$$suppr : element \times liste \rightarrow liste$$
$$suppr(e, nil) = nil$$
$$suppr(e_1, adjq(e_2, l)) = si(e_1 = e_2, suppr(e_1, l), adjq(e_2, suppr(e_1, l)))$$

3.1.3 Implantation en C

La nature même des constructeurs (*adjq*) suggère qu'un élément d'une liste permette d'accéder au suivant. De plus, il faut pouvoir étendre des listes à volonté. Tout cela implique une implatation par des pointeurs.

structure des données

Un premier type d'implantation est le suivant :

```
/* element est le type des elements de la liste, defini prealablement */
typedef struct ll1
        {
        element val;
        struct ll1 *suivant;
        } liste1;
```

Le problème d'une telle représentation est la représentation de la liste vide. Aussi préfère-t-on la modélisation suivante :

```
typedef struct ll2
        {
        liste1 *premier;
        } liste2;
```

Ainsi, la liste vide peut-être représenté par une variable de type *liste2* dont le champ *premier* est à *void*.

3.1.4 Définition des constructeurs

Attention : pour ne pas surcharger l'écriture, on a supposé disposer d'une mémoire infinie (pas d'erreur sur le malloc). Mais ceci n'est évidemment pas à faire dans un cas pratique !

```
liste2 *creer(void)
{
        liste2 *nouveau;
        nouveau = malloc(size_of(liste2));
        nouveau->premier = NULL;
        return nouveau;
}

liste2 *adjq(element e1, liste2 *ancienne_liste)
{
        liste1 *el;
        liste1 *courant;

/* creation de l'element a rajouter */
        el = malloc(sizeof(liste1));
        el->val = e1;
        el->suivant = NULL;
/* Attention : seulement si element est un type simple */
/* Sinon, il faut recopier tout l'element */
        if (!(ancienne_liste->premier))
                /* premier cas : ajout a une liste vide */
                ancienne_liste->premier = el;
        else
                /* deuxieme cas : ajout a une liste non vide */
                {
                courant = ancienne_liste->premier;
                while(courant->suivant)
                        /*On parcourt la liste jusqu'au dernier element */
                        courant = courant->suivant;
                courant->suivant = el;
```

```
                }
        return ancienne_liste
}
/* destructeur : pas un axiome, mais permet de faire le menage */
detruit(liste1 *ll)
{
        if(ll->suivant)
                detruit(ll->suivant);
        free(ll);
}
destruction (liste2 *ll)
{
        detruit(ll->premier);
        free(ll);
}
```

principaux opérateurs

- suppression en queue

```
liste2 *suppq(liste2 *ancienne_liste)
{
        liste1 *precedent, *courant;
if (ancienne_liste->premier)
                {
                courant = ancienne_liste->premier;
                if (!(courant->suivant))
                        {
                        detruit(courant);
                        ancienne_liste->premier = NULL;
                        }
                else
                        {
                        precedent = courant;
                        while(courant->suivant)
                                {
                                precedent = courant;
                                courant = courant->suivant
                                }
                        detruit(courant);
                        precedent->courant = NULL;
                        }
                }
        return ancienne_liste;
```

```
}
```

- adjonction en tête

```
liste2 *adjt(element el, liste2 *ancienne_liste)
{
        liste1 *l1;

        l1 = malloc(sizeof(liste1));
        l1->val = el;/* avec precaution */
        l1->suivant = ancienne_liste->premier;
        return (ancienne_liste->premier = l1);
}
```

- concaténation

```
liste2 *conc(liste2 *l1, *l2)
{
        liste1 *courant;
        if (courant = l1->premier)
                {
                while(courant->suivant);
                        courant = courant->suivant;
                courant->suivant = l2->premier;
                }
        else
                l1->premier = l2->premier;
        free(l2);
        return l1;
}
```

- appartient

```
int appartient(element el, liste2 *l)
{
        liste1 *courant;
        int trouve = 0;

        if (l->premier)
                {
```

```
                courant = l->premier;
                while(courant && !trouve)
                        {
                        if (courant->val == el)
                                trouve = 1;
                        courant = courant->suivant;
                        }
                }
        return trouve;
}
```

- nb_occ

```
int nb_occ(element el, liste2 *l)
{
        liste1 *courant;
        int res = 0;

        if (l->premier)
                {
                courant = l->premier;
                while(courant)
                        {
                        if (courant->val == el)
                                res++;
                        courant = courant->suivant;
                        }
                }
        return res;
}
```

- suppr_1

```
liste2 *suppr_1(element el, liste2 *l)
{
        liste1 *courant, *dernier *precedent;

        dernier = NULL;
        if (courant = l->premier)
                {
                if (courant->val == el)
                        dernier = l;
```

```
                while (courant)
                        {
                        if (courant->val == el)
                                dernier = precedent;
                        precedent = courant;
                        courant = courant->suivant;
                        }
                if (dernier)
                        {
                        courant = dernier->suivant;
                        if (dernier == l)
                                l->premier = (l->premier)->suivant;
                        else
                                dernier->suivant = (dernier->suivant)->suivant;
                        free(courant);
                        }
        return l;
}
```

- suppr

```
liste2 *suppr(element el, liste2 *l)
{
        liste1 *courant, *precedent;

        precedent = l;
        if (courant = l->premier)
                while(courant)
                        {
                        if (courant->val == el)
                                {
                                if (precedent == l)
                                        l->premier = courant->suivant;
                                else
                                        precedent->suivant = courant->suivant;
                                free(courant);
                                }
                        else
                                precedent = courant;
                        courant = courant->suivant;
                        }
        return l;
}
```

3.1.5 Variation sur les listes

Listes doublement chaînées

Pour diverses raisons, on peut créer des listes doublement chaînées : dans ces cas là, un élément de la liste possède un pointeur sur le suivant, mais aussi sur le précédent. Un avantage d'un tel procédé est que l'insertion et la suppression d'un élément de la liste se trouvent grandement facilitées. Par contre, la place prise en mémoire est plus importante.

La structure C des maillons de telles listes est alors la suivante :

```
typedef struct ll2
{
        element val;
        struct ll2 *suivant;
        struct ll2 *precedent;
} corps_de_liste_double;

typedef corps_de_liste_double *liste_double;
```

Exemple d'utilisation pour la suppression de la première occurence d'un élément :

```
liste_double suppr(element el, liste_double ll)
{
        corps_de_liste_double courant;
        liste_double ll2;

        courant = liste_double;
        while (courant)
                {
                if (courant->val == el)
                        {
                        if (courant->precedent)
                                {
                                courant->precedent->suivant = courant->suivant;
                                courant->suivant->precedent = courant->precedent;
                                free(courant);
                                return ll;
                                }
```

```
                    else
                        {
                        ll2 = courant->suivant;
                        free(courant);
                        return ll2;
                        }
                    }
                else
                    courant = courant->suivant;
                }
        return ll;
}
```

Listes circulaires

Dans le cas des listes circulaires, le dernier élément pointe sur le premier. On utilise par exemple souvent ce type de listes pour modéliser des mémoires tampons, dont la stucture n'est en fait pas celle des listes linéaires, mais des files d'attente.

3.2 Les arbres

3.2.1 Introduction

La structure d'arbre est une structure assez familiaire qui, de plus, trouve de nombreuses applications en informatique. Nous verrons par exemple qu'une telle structure accélère grandement la recherche d'un élément par rapport à la recherche dans une liste chaînée.

Un arbre est soit une *feuille* (étiquetée), soit un *nœud interne*, qui, outre une valeur (s'il est étiqueté), possède plusieurs *fils*, qui sont eux-mêmes des arbres. De plus, un arbre, possède un nœud qui n'a pas de *père*, appelé *racine* de l'arbre. Enfin, un arbre est un graphe acyclique (un nœud possède au maximum un père) et connexe (mis à part la racine de l'arbre, tout nœud possède un père).

On appelle arbre binaire un arbre dont tout nœud interne feuille possède exactement deux fils. On appelle arbre unaire-binaire un arbre dont tout nœud interne possède au plus deux fils.

3.2.2 Type abstrait : exemple des arbres binaires étiquetés

comme pour les listes on dispose, pour la *sorte* "arbre binaire" de constructeurs. N.B. : dans la représentation choisie, on supposera qu'une feuille est l'arbre de base. Dans d'autres représentations, on pourrait envisagé un *arbre vide*, et les feuilles seraient alors des nœuds dont les deux fils seraient des arbres vides.

- feuille(el) : element $\rightarrow$ arbre
- cons(el, a1, a2) : element $\times$ arbre $\times$ arbre $\rightarrow$ arbre

On peut alors définir quelques opérateurs sur les arbres. par exemple :

- calcul de la profondeur maximale de l'arbre ;
- calcul de la taille de l'arbre en nombre de nœuds (nœuds internes ou feuilles) ;
- extraction des fils gauche et droit ;
- obtention de la valeur de la racine de l'arbre ;
- recherche d'un élément dans un arbre non-ordonné ;
- ajout d'un élément à un arbre *ordonné* pour conserver l'ordre ;
- ...
- $prof : arbre \rightarrow nat$

$$prof(feuille(el)) = 1$$
$$prof(cons(el, a_1, a_2)) = \max(prof(a_1), prof(a_2)) + 1$$

- $taille : arbre \rightarrow nat$

$$taille(feuille(el)) = 1$$
$$taille(cons(el, a_1, a_2)) = taille(a_1) + taille(a_2) + 1$$

- $fils\_g : arbre \rightarrow arbre$

$$fils\_g(feuille(el)) = \vdash$$
$$fils\_g(cons(el, a_1, a_2)) = a_1$$

- $fils\_d : arbre \rightarrow arbre$

$$fils\_d(feuille(el)) = \vdash$$
$$fils\_d(cons(el, a_1, a_2)) = a_2$$

- $val\_rac : arbre \rightarrow element$

$$val\_rac(feuille(el)) = el$$
$$val\_rac(cons(el, a_1, a_2)) = el$$

- $recherche : element \times arbre \rightarrow bool$

$$recherche(el_1, feuille(el_2)) = (el_1 = el_2)$$
$$recherche(el_1, cons(el_2, a_1, a_2)) = \left| \begin{array}{l} (el_1 = el_2) \vee \\ recherche(el_1, a_1) \vee \\ recherche(el_2, a_2) \end{array} \right.$$

- $ajouter : element \times arbre \rightarrow arbre$

$$ajouter(el_1, feuille(el_2)) = \begin{cases} \text{si } el_1 = el_2 \text{ alors } feuille(el_2) \\ \text{si } el_1 < el_2 \text{ alors } cons(el_2, feuille(el_1), feuille(el_2)) \\ \text{sinon } cons(el_1, feuille(el_2), feuille(el_1)) \end{cases}$$

$$ajouter(el_1, cons(el_2, a_1, a_2)) = \begin{cases} \text{so } el_1 = el_2 \text{ alors } cons(el_2, a_1, a_2) \\ \text{si } el_1 > el_2 \text{ alors } cons(el_2, a_1, ajouter(el_1, a_2)) \\ \text{sinon } cons(el_2, ajouter(el_1, a_1), a_2) \end{cases}$$

3.2.3 Implantation en C des arbres unaires-binaires

Histoire de varier les plaisirs, la structure des arbres implantés ici n'est pas la même que celle exposée dans la partie précédente.

Comme pour les listes, l'implantation des arbres en C se fait aux moyens de structures, de pointeurs, et de l'allocation dynamique. Par mesure de simplicité, on considérera que des feuilles sont des nœuds sans fils.

structure de base

```
/* element est le type des elements de l'arbre, defini prealablement */
typedef struct tree
        {
        element val;
        struct tree *fils_gauche;
        struct tree *fils_droite;
        } arbre;
```

---

constructeurs

```
arbre *creer_feuille(element el)
{
        arbre *bebe;

if (bebe = malloc(size_of(arbre)))
                {
                bebe->val = el;
                bebe->fils_gauche = NULL;
                bebe->fils_droit = NULL;
                }
        return bebe;
}

arbre *cons(element el, arbre *a_g, arbre *a_d)
{
        arbre *pepe;

if (pepe = malloc(size_of(arbre)))
                {
                pepe->val = el;
                pepe->fils_gauche = a_g;
                pepe->fils_droit = a_d;
                }
        return pepe;
}
```

Quelques opérateurs

```
int profondeur_max(arbre *ar)
{
        int p1, p2;

        if (ar == NULL)
                return 0;
        p1 = profondeur(ar->fils_gauche);
        p2 = profondeur(ar->fils_droit);
        if (p1 > p2)
                return 1 + p1;
        else
                return 1 + -2;
        }

int taille(arbre *ar)
{
        int p1, p2;

        if (ar == NULL)
                return 0;
        p1 = profondeur(ar->fils_gauche);
        p2 = profondeur(ar->fils_droit);
        return 1 + p1 + p2;
}

int recherche(element el, arbre *ar)
{
        if (el == ar->val)
return 1;
        else if (el < ar->val)
                return recherche(element, ar->fils_gauche);
        else
                return recherche(element, ar->fils_droit);
}

arbre *ajouter(element el, arbre *ar)
{
        arbre *ss_ar;
        arbre *nvl_ar;

        if (el == ar->val)
                return ar;
```

```
        if (el < ar)
                ss_ar = ar->fils_gauche;
        else
                ss_ar = ar->fils_droit;
        if (ss_ar == NULL)
                {
                nvl_arbre = malloc(size_of(arbre));
                }
        else
                {
                ajouter(el, ss_ar);
                return ar;
                }
}

arbre *suppr_min(arbre *ar)
{
        arbre *courant, *pere;

        pere = NULL;
        courant = ar;
        while (courant->fils_gauche)
                {
                pere = courant;
                courant = courant->fils_gauche;
                }
        if (pere)
                {
                return courant;
                pere->fils_gauche = NULL;
                }
        else
                return NULL;
}

arbre *suppr(element el, arbre *ar)
{
        arbre *aa;

        if (ar == NULL)
                return NULL;
        if (el < ar->val)
                {
                ar->fils_gauche = suppr(el, ar->fils_gauche);
                return ar;
                }
```

```
        else if (el > ar->val)
                {
                ar->fils_droit = suppr(el, ar->fils_droit);
                return ar;
                }
        else
                {
                if (ar->fils_droit == NULL && ar->fils_gauche == NULL)
                        {
                        free(ar);
                        return NULL;
                        }
                if (ar->fils_gauche == NULL)
                        {
                        aa = ar->fils_droit;
                        free(ar);
                        return aa;
                        }
                if (ar->fils_droit == NULL)
                        {
                        aa = ar->fils_gauche;
                        free(ar);
                        return aa;
                        }
                aa = suppr_min(ar->fils_droit);
                aa->fils_gauche = ar->fils_gauche;
                aa->fils_droit = ar->fils_droit;
                free(ar);
                return aa;
                }
}
```

3.2.4 Les arbres n-aires

Introduction

On peut avoir besoin de générer des arbres dont chaque nœud possède un nombre variable (*a priori* non borné) de fils. Dans ce cas, au lieu d'avoir une structure avec un pointeur par fils, on aura un pointeur sur une liste chaînée dont chaque maillon sera un fils.

En règle général, on appelle une telle liste chaînée d'arbres une *forêt*.

Implantation en C

```
typedef struct arb arbre;

typedef struct cdf
{
        arbre *val;
        struct cdf *suivant;
} corps_de_foret;

typedef corps_de_foret *foret;

struct arb
{
        element val;
        foret enfants;
};
```

3.2.5 arbres équilibrés

Quelques définitions

On appelle *facteur d'équilibre* d'un arbre la différence de hauteur entre son sous-arbre gauche et son sous-arbre droit.

Un arbre est dit *parfaitement équilibré* si toutes les branches ont même longeur.

Un arbre est dit *équilibré* si :

- son facteur d'équilibre appartient à $\{0, +1, -1\}$;
- chacun de ses sous-arbres est équilibré.

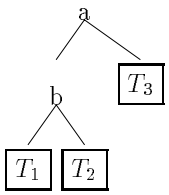
On recherche fréquemment à avoir des arbres de recherche équilibrés. En effet, cela permet d'obtenir un temps moyen de recherche optimal, de l'ordre, pour un arbre unaire-binaire, de $\log_2(n)$, où n est le nombre d'éléments de l'arbre.

maintenir un arbre équilibré

Lorsque l'on rajoute un élément à un arbre équilibré, on peut rompre cet équilibre. La méthode suivante permet, en un faible nombre d'étape, de rétablir l'équilibre rompu.

On distingue 4 cas, après ajout d'un élément à un arbre équilibré :

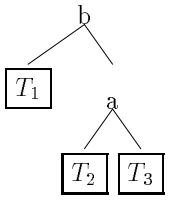
1. on obtient un arbre de la forme suivante (facteur +2, facteur gauche : +1) :



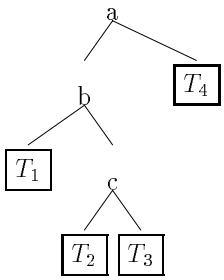
Où :

- T_1 est un sous-arbre de profondeur $h+1$;
- T_2 est un sous-arbre de profondeur h ;
- T_3 est un sous-arbre de profondeur h ;

Alors on *transforme* cet arbre en l'arbre suivant :



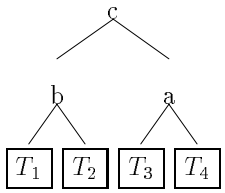
2. arbre de la forme suivante (facteur +2, facteur gauche -1) :



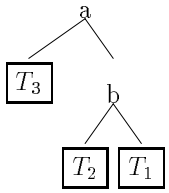
Où :

- T_1 est un sous-arbre de profondeur h ;
- $cons(c, T_2, T_3)$ est un sous-arbre de profondeur $h+1$;
- T_4 est un sous-arbre de profondeur h ;

Alors on *transforme* cet arbre en l'arbre suivant :



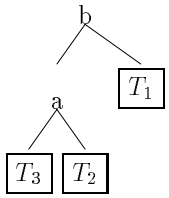
3. on obtient un arbre de la forme suivante (facteur -2, facteur droit : -1) :



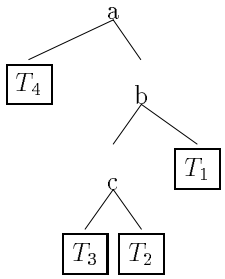
Où :

- T_1 est un sous-arbre de profondeur $h+1$;
- T_2 est un sous-arbre de profondeur h ;
- T_3 est un sous-arbre de profondeur h ;

Alors on *transforme* cet arbre en l'arbre suivant :



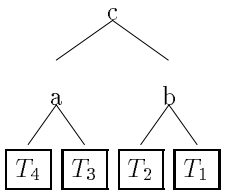
4. arbre de la forme suivante (facteur -2, facteur droit +1) :



Où :

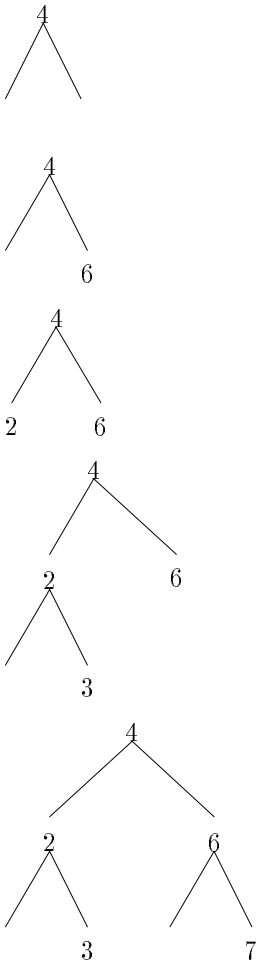
- T_1 est un sous-arbre de profondeur h ;
- $cons(c, T_3, T_2)$ est un sous-arbre de profondeur $h+1$;
- T_4 est un sous-arbre de profondeur h ;

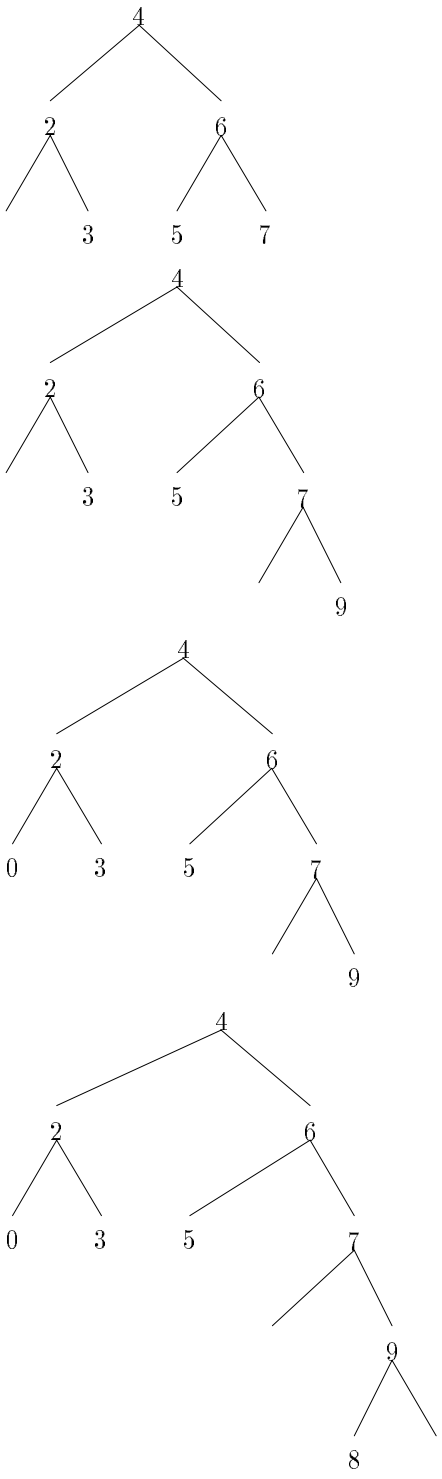
Alors on *transforme* cet arbre en l'arbre suivant :

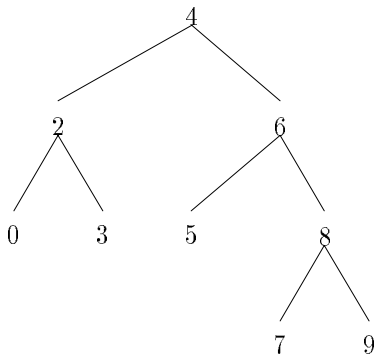


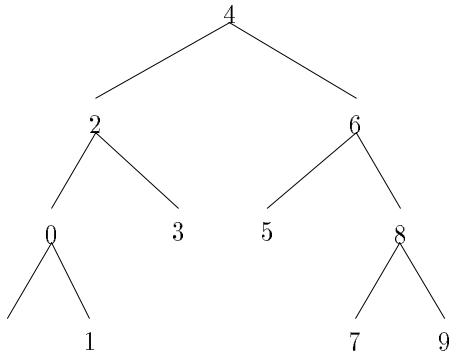
Exemple

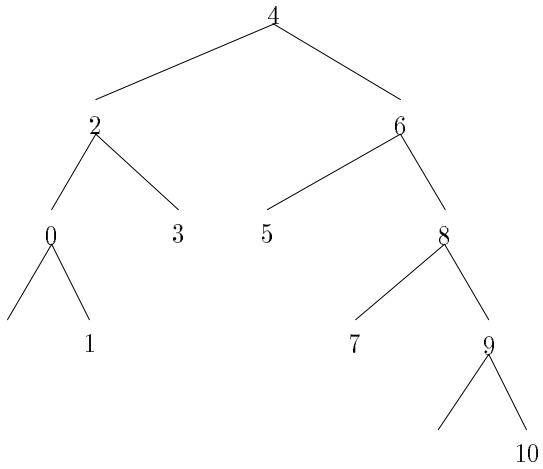
Dans l'exemple qui suit, on construit un arbre qui reste équilibré, en ajoutant successivement les nombres suivants : 4, 6, 2, 3, 7, 5, 9, 0, 8, 1 et 10.

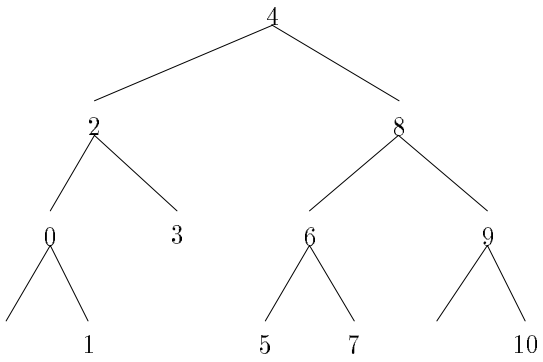



4
2
6
3
5
7
4
2
6
3
5
7
9
4
2
6
0
3
5
7
9
4
2
6
0
3
5
7
9
8


4
2
6
0
3
5
8
7
9


4
2
6
0
3
5
8
1
7
9


4
2
6
0
3
5
8
1
7
9
10


4
2
8
0
3
6
9
1
5
7
10

3.3 Les graphes

3.3.1 Introduction

Présentation

On appelle *graphe orienté* un système composé d'un ensemble de sommets $\mathcal{N}$ et d'une relation binaire $\mathcal{A}$ entre ces sommets.

Les graphes permettent de représenter un vaste ensemble de données en tout genre. Cela peut être des trajets, des propriétés mathématiques transitives telles que l'inclusion, etc.

Quelques définitions

On appelle *arc* toute paire de $\mathcal{A}$.

Si (s_1, s_2) est un arc, alors s_1 est l'*extrémité initiale* de l'arc, et s_2 l'*extrémité terminale*.

On appelle *successeur* d'un sommet s tout sommet t tel que $(s,t) \in \mathcal{A}$, et on appelle *prédécesseur* d'un sommet s tout sommet t tel que $(t,s) \in \mathcal{A}$.

On appelle *chemin* de longueur $p-1$ dans un graphe orienté toute liste de sommets $(n_1, \ldots, n_p)$ telle que :

$$\forall i \in \{1, \ldots, p-1\}, (n_i, n_{i+1}) \in \mathcal{A}$$

On appelle *circuit élémentaire* dans un graphe orienté un chemin $(n_1, \ldots n_p)$ tel que :

$$\begin{cases} n_1 = n_p \\ \nexists (i,j) \in \{1, \ldots, p\}^2 \text{ t.q. } (i,j) \leq (1,p) \wedge i < j \wedge n_i = n_j \end{cases}$$

On appelle *cycle élémentaire* une suite (n_1, n_p) de sommets tels que :

$$\begin{cases} n_1 = n_p \\ \forall i \in \{1, \ldots, p-1\}, (n_i, n_{i+1}) \in \mathcal{A} \vee (n_{i+1}, n_i) \in \mathcal{A} \\ \nexists (i,j) \in \{1, \ldots, p\}^2 \text{ t.q. } (i,j) \leq (1,p) \wedge i < j \wedge n_i = n_j \end{cases}$$

Un graphe orienté qui possède un cycle est dit *cyclique*. Dans le cas contraire, il est *acyclique*.

Un graphe *non orienté* est un graphe dans lequel la relation $\mathcal{A}$ est symétrique.

Dans un graphe non-orienté, on peut définir la connexité comme relation d'équivalence. Deux sommets a et b sont connexes s'il existe un chemin de a vers b. Une classe d'équivalence de cette relation est appelée *composante connexe*.

3.3.2 Représentation des graphes

Il existe de nombreuses manières pour représenter les graphes. En voici quelques unes :

1. Listes de nœuds et d'arêtes : On peut représenter les graphes tels qu'ils sont décrits mathématiquement : on dispose de deux types de données :

une liste de nœuds et une liste d'arêtes. Avec une telle représentation, la recherche des successeurs d'un nœud nécessite de parcourir toute la liste d'arête (où une partie si elle est triée). Cette solution est donc en général peut efficace, et n'est donc pas beaucoup utilisée.

2. Listes d'adjacences : Dans cette représentation, on dispose d'une liste de couples dont le premier élément est un nœud et le deuxième une liste des successeurs de ce nœud.

3. Matrice d'incidence : Si l'on a n nœuds, on dispose d'une matrice M de taille $n \times n$ à valeur dans $\mathbb{B} = \{vrai, faux\}$, , telle que $M[p,q] = vrai \Leftrightarrow q$ est un successeur de p. Cette méthode permet notamment d'utiliser le calcul matriciel pour les relations d'adjacence et de connexité.

3.3.3 Implantation en C

Parmi les multiples modèles de représentation présentés ci-dessus, nous choisissons ici une représentation sous forme matricielle. La représentation sous forme d'un tableau présentée ici offre bien sûr des avantages et des inconvénients : elle est efficace, aussi bien en terme d'espace mémoire utilisé qu'en terme de rapidité des algorithmes implantés, mais elle n'est pas aisément extensible. Cependant, l'adjonction de sommets dynamiquement dans un graphe étant rarement pratiquée, cette représentation est la plus fréquemment utilisée.

Deux options existent encore, suivant que l'on considère qu'un nœud est relié à lui-même ou non. Ici, on supposera que non. Cela permet notamment de traiter des graphes dans lesquels certains nœuds peuvent être leur propre successeur.

```
#define T 100
typedef int sommet;
typedef sommet graphe[T][T];
typedef Arc
{
        sommet origine;
        sommet destination;
} arc;

void init(graphe g)
{
        int i, j;
        for (i = 0 ; i < T ; i++)
                for(j = 0 ; j < T ; j++)
                        g[i][j]= 0;
}
```

```
int ajouter_arete(graphe g, arc *a)
{
        if (a->origine >= T || a->destination >= T)
                return 0;
        if (g[a->origine][a->destination])
                return 0;
        return (g[a->origine][a->destination] = 1);
}

int supprimer_arete(graphe g, arc *a)
{
        if (a->origine >= T || a->destination >= T)
                return 0;
        if (!g[a->origine][a->destination])
                return 0;
        g[a->origine][a->destination] = 0;
        return 1;
}
```

3.3.4 Recherche des composantes connexes

Principe

Si G_1 est la matrice d'incidence du graphe $\mathcal{G}$, alors la matrice $G_2 = G_1 \times G_1$ est telle que :

- $G_2[i][j] = 0$ si on ne peut aller de i à j en exactement deux étapes ;
- $G_2[i][j] \neq 0$ sinon.

En itérant le processus au plus $n-1$ fois, où n est le nombre de sommets du graphe, et en faisant la somme des matrices obtenues successivement, on trouve la *fermeture transitive* du graphe. Si l'on *symétrise* le graphe, et qu'on rajoute les relations de réflexivité, on obtient, ligne par ligne, les composantes connexes.

Implantation

```
#define min(X,Y) (((X) < (Y))? (X) : (Y))

void calc_composantes_connexes(graphe gs, graphe gd)
{
        int i,j,k,l;
```

```
        graphe gtemp;

        for(i = 0; i < T ; i++)
                for(j = 0 ; j < T ; j++)
                        gd[i][j]=gtemp[i][j]=gs[i][j];

        for (l = 1; l < T ; l++)
                {
                for(i = 0 ; i < T ; i++)
                        for(j = 0 ; j < T ; j++)
                                {
                                for(k = 0 ; k < T ; k++)
                                        gd[i][j]+=gtemp[i][k]*gs[k][j];
                                gd[i][j] = min(gd[i][j],1);
                                }
                for (i = 0 ; i < T ; i++)
                        for (j = 0; j < T ; j++)
                        ¿gtemp[i][j]=gd[i][j];
                }
}

void composantes_connexes(graphe g)
{
        graphe g1, g2;
        sommet crible[T];
        int i,j;

        for (i = 0; i < T ; crible[i++]=1);
        for (i = 0 ; i < T ; i++)
                for (j = 0 ; j < T ; j ++)
                        g1[i][j] = g[i][j] || g[j][i] || (i==j);

        calc_composantes_connexes(g1, g2);

        for (i = 0 ; i < T ; i++)
                if (crible[i])
                        {
                        for(j = i ; j < T ; j++)
                                if (g2[i][j])
                                        {
                                                printf("%d\t",j);
                                                crible[j]=0;
                        }
                        printf("\n");
                        }
}
```

3.3.5 Recherche de chemin dans un graphe décoré

Principe

tout comme les arbres, les graphes peuvent être *décorés*, c'est-à-dire que les nœuds n'ont pas seulement un identificateur, mais aussi une valeur. Là encore, plusieurs types de représentation peuvent être choisis : on peut par exemple avoir un tableau qui à un sommet associe une valeur.

Pour rechercher si un nœud dont la valeur possède une certaine propriété est accessible depuis un nœud donné n, il existe plusieurs types d'algorithme :

- recherche en *profondeur d'abord* : on part de n, et on suit un premier chemin jusqu'au bout, c'est-à-dire jusqu'à ce que l'on tombe sur un nœud sans successeur ou un nœud dont tous les successeurs ont déjà été visités. Si l'on n'a toujours pas trouvé le nœud recherché, on revient en arrière d'un pas, et on essaie un autre chemin.
- recherche en *largeur d'abord* : de n, on regarde tous ses successeurs. Si aucun d'eux n'est le nœud recherché, alors on regarde leurs successeurs. On continue ainsi de suite jusqu'à tomber sur le nœu recherché où jusqu'à avoir parcouru tout le graphe.

Implantation en C

La solution présentée ici est la solution en profondeur d'abord. Remarque : l'algorithme est donné sur un arbre non décoré pour alléger l'écriture.

```
int recherche_rec(graphe g, sommet depart, sommet final, sommet visite[])
{
        int i;
                for (i = 0; i < T; i++)
                {
                if (g[depart][i] && !visite[i])
                        {
                        visite[i]=1;
                        if (i == final)
                                return 1;
                        else if (recherche_rec(g, i, final, visite))
                                return 1;
                        }
                }
}
```

```
int recherche(graphe g, sommet depart, sommet final)
{
        sommet visite[T];
        int i;

        for (i = 0 ; i < T ; visite[i++]=0);
        return recherche_rec(g, depart, final, visite);
}
```

3.3.6 graphe pondéré

Introduction

Un *graphe pondéré* est un graphe dont les arêtes sont affectées d'un poids. Cela permet d'utiliser les graphes pour représenter, par exemple, des plans.

Pour la représentation de tels graphes, on peut continuer à utiliser les matrices d'incidence, dont les composantes représentent les poids des arcs. Le produit matriciel n'a alors plus de sens. Cependant, on peut continuer à l'utiliser pour les problèmes de connexité, si l'on garde le zéro pour indiquer qu'il n'y a pas d'arrête (ce qui implique qu'il n'y a pas d'arc de poids nul).

Arbre de recouvrement

Un arbre de recouvrement d'un graphe connexe $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ est un traphe $\mathcal{T} = (\mathcal{N}_1, \mathcal{A}_1)$ tel que :

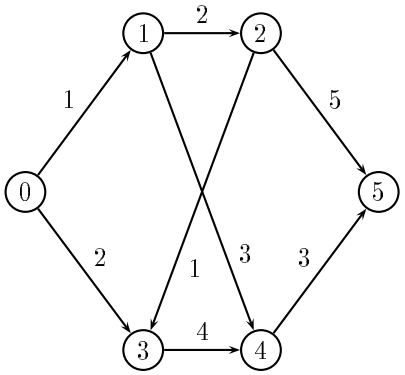
- $\mathcal{A}_1 \in \mathcal{A}$
- $\mathcal{T}$ n'a pas de cycle
- $\mathcal{N} = \mathcal{N}_1$
- $\mathcal{T}$ est connexe

Un arbre de recouvrement est *minimal* si pour tout autre arbre de recouvrement, la somme des coûts des arrêtes le composant est supérieure ou égale.

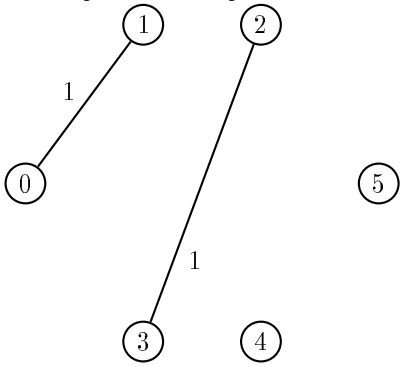
Algorithme de Kruskal

Il s'agit d'un algorithme qui permet, à partir d'un graphe connexe, de construire un arbre de recouvrement minimal. Le principe de l'algorithme est le suivant : on prend les arcs par ordre croissant de poids, et l'on ne conserve que les arcs qui ne relient pas deux nœuds reliés à des arrêtes précédemment conservées.

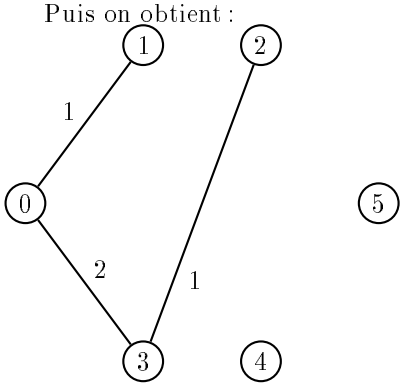
Exemple : soit le graphe suivant :



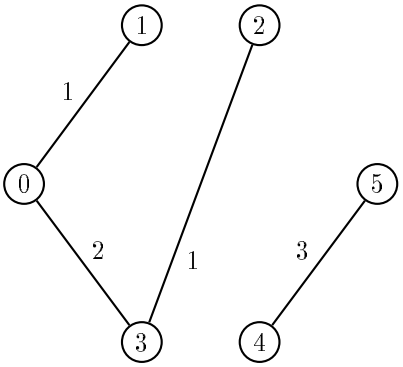
La première étape nous donne :



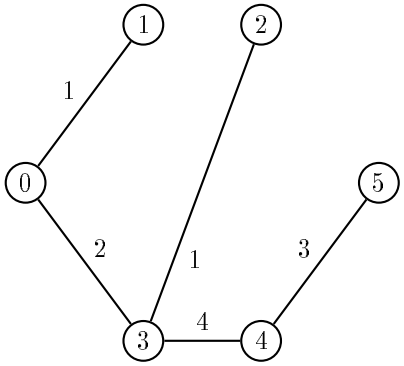
Puis on obtient :



À l'étape suivante, on a:



Pour enfin se trouver avec :



Recherche du plus court chemin

Étant donné deux sommets d'un graphe, on peut rechercher le "plus cours chemin" (celui dont la somme des poids des arcs est minimale) joignant le premier de ces sommets au deuxième. Pour cela, on peut procéder par un parcours en largeur d'abord ou en profondeur d'abord. Une autre technique est la technique des flots : on propage de sommet en sommet la distance depuis l'origine. Une fois le nœud destination atteint, il suffit de remonter en sélectionnant à chaque fois le sommet d'où l'on venait. Cette technique limite beaucoup les étapes récursives inutiles.

```
typedef struct Noeud
{
        int distance;
        int provenance;
} noeud;

void affiche_chemin(noeud tableau[T], sommet i)
{
        if (tableau[i].provenance == -1)
                printf("origine : %d\n", i);
        else
```

```
                {
        affiche_chemin(tableau, tableau[i].provenance);
        printf("sommet %d\n", i);
                }
}

void plus_court_chemin_rec(graphe g, sommet origine, sommet destination, noeud tableau[T])
{
        int i, dist;

        for (i = 0 ; i < T ; i++)
                {
                if (g[origine][i])
                        {
                        /* Il existe une arrete entre origine et i */
                        dist = tableau[origine].distance + g[origine][i];
                        if (    (tableau[i].provenance == -1)
                                /* C'est la premiere fois qu'on visite ce noeud */
                        ||      (tableau[i].provenance != -1 && dist < tableau[i].distance))
                                /* Noeud deja visite, mais on a trouve un chemin plus court */
                                {
                                tableau[i].distance = dist;
                                tableau[i].provenance = origine;
                                plus_court_chemin_rec(g, i, destination, tableau);
                                }
                        }
                }
}

plus_court_chemin(graphe g, sommet origine, sommet destination)
{
        noeud tableau[T];
        int i;

        for (i = 0 ; i < T ; tableau[i++].provenance = -1);

        plus_court_chemin_rec(g, origine, destination, tableau);

        if (tableau[destination].provenance == -1)
                printf("pas de chemin\n");
        else
                {
                affiche_chemin(tableau, tableau[destination].provenance);
                printf("destination : %d\n", destination);
                }
}
```

3.4 Les files

3.4.1 Introduction

Les files sont des structures de type premier-entré/premier-sortie (en anglais, FIFO : First In, First Out). Les mémoires tampons (buffers), les *pipes*, notamment, sont de ce type.

On dispose de deux créateurs :

- $nil : \rightarrow file$
- $adjq : file \times element \rightarrow file$

Il existe aussi deux fonctions d'accès :

- lecture de l'élément de tête : hd
- suppression de l'élément de tête : tl

Ceux-ci sont définis ainsi :

$$
\begin{aligned}
&hd : file \rightarrow element \\
&hd(nil) = \perp \\
&hd(adjq(nil, el)) = el \\
&hd(adjq(adjq(l, el_1), el_2)) = hd(adjq(l, el_1))
\end{aligned}
$$

$$
\begin{aligned}
&tl : file \rightarrow file \\
&tl(nil) = \perp \\
&tl(adjq(nil, el)) = nil \\
&tl(adjq(adjq(l, el_1), el_2)) = adjq(tl(adjq(l, el_1)), el_2)
\end{aligned}
$$

3.4.2 Implantation en C

En général, les files utilisées en informatique font simultanément les opérations *hd* et *tl*. De plus, elles ont une capacité limitée. Elles peuvent donc être représentées par un tableau. On a alors deux problèmes :

- il faut mémoriser l'endroit où le prochain élément doit être mis ;
- chaque fois que l'élément de tête est "consommé", il faut décaler tout le tableau, ce qui peut être très long.

Le premier point implique donc d'utiliser non pas un simple tableau, mais une structure comportant un tableau et un indice. Le deuxième problème peut être résolu grâce à l'opérateur *modulo* : au lieu de décaler le tableau, on modifie le pointeur de tête de la file sur l'élément suivant. La taille du tableau étant limitée, il faut, arriver à la fin du tableau, recommencer à partir du début, la

partie située avant l'indice de tête étant en fait libre. Il faut donc avoir de plus dans la structure un indice pour le premier élément courant.

```
#define TAILLE 100
#define T (TAILLE + 1)
typedef struct fl
{
        int tete;
        int next;
        element F[T];
} file;
init(file *f)
{
        f->tete = 0;
        f->next = 0;
}

file *prod(file *f, element el)
{
        if (f->tete - f->next == -1 ||
                (f->tete == 0 && f->next == TAILLE))
                return NULL;
        f->F[f->next] = el;
        f->next = (f->next+1) % T;
        return f;
}

element cons(file *f, element el)
{
        element e ;

        if (f->tete == f->next)
                /* on suppose que la file ne peut pas contenir */
                /* l'element -1 */
                return -1;
        e = f->F[f->tete];
        f->tete = (f->tete + 1) % T;
        return e;
}
```

Une autre solution pour implanter les files est d'utiliser les listes circulaires (voir paragraphe 3.1.5).

3.5 Les piles

3.5.1 Introduction

Les piles sont des structures de types dernier-entré/premier-sortie (En anglais, LIFO : Last In, First Out). Les piles sont beaucoup utilisées en informatique dans les langages de bas niveau (assembleur). C'est aussi un moyen commode pour dé-récursiver un problème.

Les constructeurs sont semblables à ceux que l'on trouve pour les files :

- $nil : \rightarrow pile$

- $push : pile \times element \rightarrow pile$

On trouve notamment les opérateurs suivants :

- $pop : pile \rightarrow pile$

- $pull : pile \rightarrow element$

définis ainsi :

$$pop(nil) = \perp$$
$$pop(push(p, e)) = p$$

$$pull(nil) = \perp$$
$$pull(push(p, e)) = e$$

3.5.2 Implantation en C

Comme pour les files, la taille d'une pile est en général limitée. De plus, en théorie, les opérations *pop* et *pull* sont souvent simultanées. En pratique, l'utilisation d'une pile est plus souple. Notamment, on peut accéder aux i-ème élément avant le somment de pile.

```
#define TAILLE 100
typedef struct pl
{
        int sommet;
        element P[TAILLE+1];
} pile;

init(pile *p)
{
```

```
        p->sommet = 0;
}

pile *empile(pile *p, element el)
{
        if (p->sommet > TAILLE)
                return 0;
        p->P[p->sommet++]=el;
        return p;
}
element depile(pile *p)
{
        if (sommet == 0)
                /* on suppose que -1 ne peut etre dans la pile */
                return -1;
        return p->P[p->sommet-];
}
element acces(pile *p, int delta)
{
        int indice;

        if (p->sommet - delta < 0)
                return -1;
        return (p->P[indice]);
}
```

Chapitre 4

Algorithmes fondamentaux

4.1 Recherche dans un ensemble de données trié

4.1.1 Introduction

Il y a principalement deux raisons pour rechercher un élément dans une liste :

- savoir si l'élément est présent ;
- connaître des informations sur un élément dont on connaît la clef (recherche associative)

La similarité des algorithmes de recherche fait qu'ici, on ne présentera que ceux liés au premier type.

Dans le cas de la liste simple envisagée dans une partie précédente, la seule recherche envisageable est une recherche séquentielle en partant du premier élément de la liste. En effet, on ne peut accéder à un élément de la liste qu'à partir d'un chaînage partant du premier. Le fait que la liste soit triée permet d'éviter de parcourir toute la liste : dès qu'on atteint un élément "supérieur" à l'élément recherché, on peut s'arrêter.

4.1.2 Liste chaînée : recherche séquentielle

```
int recherche(liste2 *ll, element el)
{
        liste1 *courant;
        int trouve = 0;

        if (courant = ll->premier)
                {
                /* Liste non vide */
                while (courant && !trouve)
                        if (courant->el == el)
                                trouve = 1;
                        else if (courant->el ¿el)
                                courant = NULL;
                }
        return trouve;
}
```

Complexité de l'algorithme (en moyenne) : $O(n)$

4.1.3 Recherche dans un tableau trié

Dans le cas des tableaux, on peut accéder directement au i-ème élément. Ceci permet d'utiliser des algorithmes plus rapides, comme la recherche par

dichotomie.

```
int recherche_dicho (int *tableau, int taille, element el)
{
        int trouve = 0;
        int min, max, milieu;

        min = 0;
        max = taille-1;

        if (tableau[min] == el || tableau[max]==el)
                return (trouve = 1);

        while (max-min > 1 && !trouve)
                if(tableau[milieu = (min + max) / 2] == el)
                        trouve = 1;
                else if (tableau[milieu]¿el)
                        max = milieu;
                else
                        min = milieu;
        return trouve;
}
```

Complexité de l'algorithme (au pire) : $O\ (log_2\ (n))$

Cet algorithme est donc bien plus rapide que le précédent. Par exemple, pour 1000 données, le premier prendra en moyenne 500 coups. Celui-ci, dans le pire des cas, prendra 10 coups. Aussi apparaît-il intéressant de trouver des moyens pour accélérer la recherche dans le premier cas.

4.1.4 Table de hachage

Le principe d'une table de hachage est de garder un accès direct sur certains éléments d'une liste triée dans un tableau, où dans une autre liste (la table de hachage). Lorsque l'on recherche un élément, on commence par le comparer avec les éléments servant de clef dans la table de hachage. Puis on lance la recherche dans la liste principale en commençant au plus grand élément de la table de hachage inférieur à l'élément recherché.

Une table hachage peut être statique (par exemple, pour un dictionnaire, une entrée pour chaque lettre de l'alphabet) ou dynamique (on rajoute des entrées dans la table au fur et à mesure des besoins). Dans ce dernier cas, lorsque l'on constate qu'il y a trop de données entre deux entrées de la table de hachage, on crée une nouvelle clef qui pointera sur l'element du milieu. Tandis que dans le premier cas, la table de hachage peut être implantée par un tableau, dans le deuxième cas, il faudra passer par une liste chaînée. Celle-ci peut d'ailleurs aussi être gérée par une table de hachage ! On verra par la suite qu'une autre structure que la liste chaînée (les arbres) peut grandement accélérer la recherche.

4.2 Les tris

Dans le cas où l'on dispose d'un tableau de données, il peut être intéressant de pouvoir trier ce tableau. Cela permet notamment d'accélérer par la suite la recherche d'éléments dans un tableau (Imaginez un dictionnaire dans lequel les mots ne seraient pas triés par ordre alphabétique !).

Les algorithmes présentés ici seront donnés sur les tableaux, mais ils s'appliquent aussi dans le cas où les données sont stockées sous forme de liste chaînée. Le parcours est par contre légèrement plus compliqué.

4.2.1 Tri par insertion

Dans le cas où les données n'ont qu'un ordre pertinent, et si l'ajout de nouvelles données est rare, il est plus judicieux de mettre directement une nouvelle donnée à sa place, plutôt que de la mettre à la fin, puis de tout retrier. Cela est d'autant plus vrai si les données sont implantées sous la forme d'une chaînée. C'est ce qu'on appel le tri par insertion.

Le principe, dans le cas d'une liste, est le suivant : on parcours la liste jusqu'a tomber sur un element qui est situé *après* (dans l'ordre choisi) l'élément à insérer. On revient alors en arrière d'un cran. L'élément courant devra alors pointer sur le nouvel élément, qui pointera lui-même sur l'élément sur lequel on s'était arrêté en parcourant la liste. Les cas où l'élément à rajouter doit venir en première ou dernière position sont à traiter séparément.

Dans le cas d'un tableau, il faut, après avoir trouver où devait être inséré le nouvel élément, décaler toute la deuxième partie du tableau, en partant de la fin.

Si les données ne peuvent être maintenues triées, il faut alors procéder à un remaniment de tous le tableau. C'est le rôle des algorithmes qui suivent.

4.2.2 Tri par sélection

Principe

Pour ce tri, on recherche le minimum dans le tableau, et on vient l'échanger avec le premier élément du tableau. Puis l'on recherche le suivant, que l'on échange avec le deuxième élément, etc.

Algorithme

```
void tri_par_selection(int *tableau, int taille)
{
        int min;
        int ind_min;
        int i, j;
        int temp;
```

```
        for (i = 0; i < taille-1;i++)
                {
                min = tableau[i];
                ind_min = i;
                for(j=i+1; j < taille;j++)
                        if (tableau[j] < min)
                                {
                                min = tableau[j];
                                ind_min = j;
                                }
                if (i != j)
                        {
                        temp=tableau[ind_min];
                        tableau[ind_min]=tableau[i];
                        tableau[i]=temp;
                        }
                }
}
```

4.2.3 Le tri à bulle

Principe

- Première version : Ce type de tri consiste à parcourir le tableau à partir du début jusqu'à trouver deux éléments qui ne sont pas dans l'ordre désiré. On les inverse alors et on recommence. Le processus s'arrête lorsqu'on a parcouru tout le tableau sans avoir à effectuer d'échange.

- Deuxième version : On parcourt tout le tableau jusqu'au dernier rang en inversant tous les couples d'éléments consécutifs non ordonnés. Ceci a pour effet de mettre à la fin du tableau le plus grand élément, et de pré-ordonner le reste. Puis on recommence en s'arrêtant un rang avant. Le processus continue jusqu'à ce que le tableau soit entièrement trié.

Algorithme

- Première version :

```
void tri_a_bulle_1(int *tableau, int taille)
{
        int indice=0 ;
        int temp;

        while (indice <taille-1)
                if (tableau[indice+1] < tableau[indice])
```

```
                        {
                        temp = tableau[indice];
                        tableau[indice] = tableau[indice+1];
                        tableau[indice+1] = temp;
                        indice = 0;
                        }
                else
                        indice++;
}
```

- Deuxième version :

```
void tri_a_bulle_2(int *tableau, int taille)
{
        int max;
        int i;
        int trie = 0;
        int temp;

        for (max = taille; max > 1 && !trie ; max--)
                {
                trie = 1;
                for( i = 0 ; i < max-1 ; i++)
                        if (tableau[i+1] < tableau[i])
                                {
                                temp = tableau[i];
                                tableau[i] = tableau[i+1];
                                tableau[i+1] = temp;
                                trie = 0;
                                }
                }
}
```

4.2.4 Le tri shell

Principe

ce tri est une variante du tri à bulle : il permet de faire des échanges entre des éléments distants, et pas seulement entre deux voisins : On part d'un très grand intervalle, puis on le réduit peu à peu.

Algorithme

```
void tri_shell(int *tableau, int taille)
{
        int delta;
        int temp;
        int i;
        int fini = 1;

        delta = taille-1-1;
        while (delta > 0)
                {
                for(i = 0; i <= taille-1-delta;i++)
                        if (tableau[i] > tableau[i+delta])
                                {
                                temp = tableau[i];
                                tableau[i] = tableau[i+delta];
                                tableau[i+delta] = temp;
                                fini = 0;
                                }
                if (fini)
                        delta = (delta+1)/3;
                fini = 1;
                }
}
```

4.2.5 Tri rapide : le quick sort

Principe

On choisit un élément témoin au hasard dans le tableau. On sépare le tableau initiale en deux parties : une qui contiendra tous les éléments inférieurs strictement à l'élément témoin, l'autre tous ceux qui sont supérieurs ou égaux à l'élément témoin. Et on recommence sur chacun des deux sous-tableaux obtenus. On s'arrête quand un sous-tableau examiné ne contient que zéro, un ou deux éléments.

Algorithme

```
void tri_rapide(int *tableau, int taille)
{
        int temoin;
        int ind_inf, ind_sup;
        int temp;

        if (taille > 1)
```

```
        {
        ind_inf = 0;
        ind_sup = taille-1;
        temoin = tableau[0];
        while (ind_inf <= ind_sup)
                {
                if (tableau[ind_inf] < temoin)
                        ind_inf++;
                else
                        {
                        temp = tableau[ind_inf];
                        tableau[ind_inf]=tableau[ind_sup];
                        tableau[ind_sup] = temp;
                        ind_sup--;
                        }
                }
        tri_rapide(tableau, ind_inf);
        tri_rapide(&(tableau[ind_inf]), taille-ind_inf);
        }
}
```

4.2.6 Réutilisabilité...

Il existe une fonction de la librairie de base du C intitulée `qsort` qui... implante l'algorithme de tri rapide ! Son utilisation nécessite l'inclusion du fichier `stdlib.h`. Afin de pouvoir s'adapter à tous les cas, son utilisation est un peu lourde... Enfin, elle ne s'applique pas aux listes chaînées, mais aux tableaux uniquement.

Il faut d'abord définitir d'une fonction de tri. Celle-ci doit prendre deux paramètres et retourner un entier qui vaudra :

- 0 si les deux paramètres sont égaux ;
- 1 s'ils sont dans l'ordre décroissant ;
- -1 s'ils sont dans l'ordre croissant.

La fonction `qsort` s'utilise ainsi :

```
qsort ( (type *) tableau, taille_tableau, size_of(type),
                        fonction_de_tri)
```

où `type` est le type des éléments du tableau., et `fonction_de_tri` est la fonction définie précédemment, d'en-tête :

```
int fonction_de_tri (type *e1, type *e2).
```

4.3 Les jeux

4.3.1 Introduction

Il existe de multiples types de jeux. Cependant, la plupart se ramène à un *combat* entre deux joueurs, qui jouent chacun leur tour. L'étude théorique de ce type de jeux intéresse beaucoup les mathématiciens, mais aussi les informaticiens. En effet, des algorithmes similaires à ceux utilisés pour implanter des jeux peuvent être mis en place pour résoudre des problèmes d'optimisation (de choix, stratégies, etc.). Ici, on se limitera aux jeux suivant le principe évoqué ci-dessus.

4.3.2 Algorithme du Min-Max

Principe

Le principe est assez simple : le programme essaie tous les coups possibles, et retient celui qui arrive à la meilleure solution. Si l'on veut que le programme envisage plusieurs coup successifs, il supposera qu'à chaque coup, le joueur retient le "meilleur coup" pour lui.

Dans ce type de problèmes, il faut donc disposer :

- d'une structure de données pour représenter le jeu ;
- d'une fonction générant la liste des coups possibles ;
- d'une fonction d'évaluation d'une position ;
- d'une fonction sélectionnant le meilleur coup.

Structure de données

Celle-ci est en général assez simple : dans la plupart des jeux, il s'agira d'un tableau à deux dimension, dont le contenu des cases correspondra à la pièce qui l'occupe.

Fonction générant la liste des coups

Cette fonction, si l'on veut pouvoir l'utiliser de manière à essayer tous les coups, sans "redite", pourra, par exemple, renvoyer directement une liste chaînée de tous les coups possibles dans une position donnée. Elle doit donc prendre en paramètre un descriptif de la position et du joueur qui joue (si on veut pouvoir l'utiliser dans les deux sens), et devra retourner une *liste chaînée de coups*, structure à définir.

Fonction d'évaluation

C'est en partie dans cette fonction que réside la valeur du programme. En effet, la *solution idéale* n'est souvent pas connue de manière absolue. Il s'agit donc d'une approximation de cette solution idéale. C'est ce qu'on appelle une fonction *heuristique*.

Fonction de sélection du meilleure coup

C'est dans cette fonction que réside principalement le principe de l'algorithme Min-Max. Elle se charge d'explorer tous les coups jusqu'à une profondeur donnée, de manière récursive. Une fois cette profondeur atteinte, elle renvoit le résultat de la fonction d'évaluation sur la position obtenue. Puis, aux profondeurs inférieures, renvoie, suivant qu'il s'agit du tour de l'ordinateur ou non, la valeur maximale ou minimale de tous les coups de la profondeur suivante.

En supposant que l'on dispose des définitions suivantes :

```
typedef int position;
typedef int coup;
typedef int joueur;

typedef struct meilleur_coup
{
        int valeur;
        coup cp;
} m_c;

typedef c_l
{
        coup cp;
        suivant *c_l;
} corps_liste;

typedef corps_liste *liste;

int heuristique(position pos, joueur j)
liste coups_possibles(position pos, joueur j)
```

La fonction min-max aura une forme similaire à ce qui suit :

```
m_c *min_max(position pos, joueur j, int profondeur)
{
/* j, le joueur, vaut 0 ou 1. */
/* Si j=1, il s'agit du tour ami */
```

```
        liste ll_coups;
        m_c *meilleur, *r;
        corps_liste coup_courant;

        meilleur = malloc(size_of(m_c));
        if (profondeur == 0)
                {
                meilleur->valeur = heuristique(pos, j);
                return m_c;
                }
        else
                {
                ll_coups = coups_possibles(pos, j);
                coup_courant=ll_coups;
                while(coup_courant)
                        {
                        r = min_max(jouer_coup(pos, coup_courant->coup), 1-j, profondeur-1);
                        if ((r->valeur > meilleur->valeur && jj == 1) or
                                (r->valeur < meilleur->valeur && jj == 0))
                                {
                                free(meilleur);
                                meilleur = r;
                                }
                        else
                                free(r);
                        coup_courant=coup_courant->suivant;
                        }
                free(ll_coups);
                return meilleur;
                }
}
```

4.4 Quelques algorithmes sur les matrices

4.4.1 somme et produit de deux matrices

Histoire de commencer facile, voici comment faire la somme, puis le produit, de deux matrices :

```
#define M 10
#define N 5 somme(int a[M][N], int b[M][N], int c[M][N])
{
        int i, j;

        for (i = 0 ; i < M ; i++)
                for(j = 0 ; j < N ; j++)
                        c[i][j]=a[i][j]+b[i][j];
}

produit(int a[M][N], int b[N][M], int c[M][M])
{
        int i, j, k;

        for (i = 0 ; i < M ; i++)
                for(j = 0 ; j < M ; j++)
                        for(c[i][j] = 0,k = 0 ; k < N ; k++)
                                c[i][j]+=a[i][k]+b[k][j];
}
```

Le problème d'une telle solution est que la taille des matrices est fixes. Si l'on désire des fonctions valables pour des tableaux de taille quelconque, il faut gérer soi-même l'adressage dans le tableau. Cela est encore simple pour l'addition, l'accès aux différents éléments pouvant être effectué de façon séquentielle, mais c'est plus difficile pour le produit matriciel :

```
somme_gen(int nb_lignes, int nb_col, int *a, int *b, int *c)
{
        int i;

        for(i = 0 ; i < nb_lignes*nb_col ; i++)
                *c+i = *(a+i) + *(b+i);
}

produit_gen(int nb_lignes, int nb_col, int *a, int *b, int *c)
{
        int i,j,k;
```

```
        int decalage_a, decalage_b, decalage_c;

        for (i = 0 ; i < nb_lignes ; i ++)
                for (j = 0 ; j < nb_lignes ; i ++)
                        {
                        decalage_c = i * nb_lignes + j;
                        *(c+decalage_c)=0;
                        decalage_a = i * nb_lignes;
                        decalage_b = j;
                        for (k = 0 ; k < nb_col ; i ++)
                                {
                                /* decalage_a = i*nb_lignes+k */
                                /* decalage_b = k*nb_col+j */
                                *(c+decalage_c)+=*(a+decalage_a) * *(b+decalage_b);
                                decalage_a++;
                                decalage_b+=nb_col;
                                }
                        }
}
```

4.4.2 Méthode du pivot de Gauss

Il s'agit d'une méthode *a priori* connue qui permet de résoudre un système d'équation linéaire.

```
#define T 5
int gauss(float M[T][T+1])
{
        int i, j, k;
        int trouve;
        float coef, temp;

        for(i = 0 ; i < T-1 ; i++)
                if (M[i][i] == 0)
                        {
                        /* si le pivot est nul, on en cherche un autre */
                        trouve = 0;
                        j = i;
                        while (!trouve && j < T-1)
                                {
                                trouve = M[i][++j];
                                }
                        if (!trouve)
                                /* pas de pivot possible */
```

```
                        return 0;
                    for (k = i; k <= T ; k++)
                        {
                        /* on a trouve un nouveau pivot ; */
                        /* on echange les lignes. */
                        temp = M[i][k];
                        M[i][k] = M[j][k];
                        M[j][k] = temp
                        }
                    }
                    for(j = i+1 ; j < T ; j ++)
                        {
                        if (M[j][i] != 0)
                            {
                            coef = M[j][i]/M[i][i];
                            for(k = i ; k <= T ; k++)
                                M[j][k]-=coef*M[i][k];
                            }
                        }
    return 1;
}
```

4.5 Algorithme du simplexe

4.5.1 Introduction

Cet algorithme permet de résoudre des problèmes de maximisation d'une formule linéaire pour des variables liées par des équations et des inégalités linéaires.

Exemple : on dispose des contraintes suivantes :

$$\begin{aligned} x_1 + x_2 &\leq 4 \\ x_1 + 2x_2 &\leq 6 \\ x_1 - x_2 &\leq 5 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

Et on cherche à maximiser la fonction :

$$F_{max} = 3x_1 + x_2$$

4.5.2 Principe de l'algorithme

On commence par réécrire le système de façon à n'avoir que des inégalités, avec des variables exclusivement positives. Ceci se fait par l'ajout de nouvelles variables :

$$\begin{aligned} x_1 + x_2 + e_1 &= 4 \\ -x_1 + 2x_2 + e_2 &= 6 \\ x_1 - x_2 + e_3 &= 5 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ e_1 &\geq 0 \\ e_2 &\geq 0 \\ e_3 &\geq 0 \end{aligned}$$

Les variables e_i rajoutées doivent avoir pour coefficient la valeur $+1$. Si ce n'est pas le cas (inégalité dans l'autre sens), on peut s'en sortir en écrivant non pas e_i mais $-e_i + a_i$, avec a_i et e_i positifs, et en faisant jouer à a_i l'ancien rôle de e_i

On obtient alors le tableau suivant :

e_1	e_2	e_3	x_1	x_2	d
1	0	0	1	1	4
0	1	0	-1	2	6
0	0	1	1	-1	5
0	0	0	3	1	$F_{max} - 0$

On choisit alors une variable *entrante* et une variable *sortante*. La variable entrante est celle qui a le plus grand coefficient (le plus petit si l'on cherche

un minimum) dans la dernière ligne du tableau. Elle appartient toujours à la deuxième partie du tableau.

Ici, c'est donc x_1

Pour la variable sortante, on prend la variable de gauche dont le 1 est à la ligne qui donne la plus petite valeur positive (s'il s'agit d'un maximum) en divisant les éléments de la dernière colonne ligne par ligne avec ceux de la colonne de la variable entrante. Ici :

$$\begin{bmatrix} 4 \\ 6 \\ 5 \end{bmatrix} / \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ -6 \\ 5 \end{bmatrix}$$

Le plus petit résultat positif étant 4, à la ligne 1, la variable sortante sera donc e_1.

On obtient alors la nouvelle matrice :

x_1	e_2	e_3	e_1	x_2	d
1	0	0	1	1	4
-1	1	0	0	2	6
1	0	1	0	-1	5
3	0	0	0	1	$F_{max} - 0$

On multiplie alors la partie supérieure de cette nouvelle matrice par l'inverse B_0 de la sous-matrice de gauche. Celle-ci peut être obtenue par la méthode suivante :

- Pour les colonnes qui ne sont pas celles de la variable entrante, on met la colonne de la matrice identité correspondante ;

- Pour la colonne i de la variable entrante, en $[i, i]$, on met l'inverse du pivot, le pivot étant l'élément de la colonne rentrante de la ligne i. En $[i, j]$, avec $j \neq i$, on met $-[i, j]_{entrant}/[i, i]_{entrant}$, c'est-à-dire l'opposé du produit de l'inverse du pivot par l'élement de la j-ème ligne de la colonne entrant.

D'où, ici :

$$B_0 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

Le nouveau tableau obtenu est donc :

x_1	e_2	e_3	e_1	x_2	d
1	0	0	1	1	4
0	1	0	1	3	10
0	0	1	-1	2	1
3	0	0	0	1	$F_{max} - 0$

Enfin, par combinaison linéaire avec les lignes précédentes, on supprime de la dernière ligne le terme non-nul de la partie gauche du tableau :

x_1	e_2	e_3	e_1	x_2	d
1	0	0	1	1	4
0	1	0	1	3	10
0	0	1	-1	2	1
3	0	0	0	1	$F_{max} - 0$
0	0	0	-3	-2	$F_{max} - 12$

Si les deux valeurs de la dernière ligne du tableau précédent avaient été positives ou nulles (négatives ou nulles si l'on recherche un minimum), il aurait fallu recommencer. Ici, ce n'est pas le cas. On obtient alors que la fonction F_{max} à pour maximum 12, lorsque l'on a $x_1 = 1, e_2 = 10, e_3 = 1$. Une fois ces valeurs remplacées dans le système linéaire de départ, on obtient pour x_2 la valeur 0.

Autre exemple :

$$\begin{cases} -x_1 + x_2 + x_3 \leq 100 \\ x_1 - x_2 + x_3 \geq 10 \\ 2x_1 - x_2 - x_3 \geq 20 \end{cases}$$

avec $F_{max} = x_1 + x_2 + x_3$

4.6 Les algorithmes génétiques

4.6.1 Introduction

Il s'agit d'un type d'algorithmes d'optimisation assez particulier, dans la mesure où il est fondé sur le hasard. Le principe essaie de reproduire le système de la reproduction de deux cellules vivantes. On dispose d'une *population*, d'une taille donnée. Selon une fonction d'adaptation, les *individus* de cette population vont plus ou moins enclin à se reproduire. Cependant, des mutations peuvent se produire. Enfin, lors de la reproduction, il peut y avoir un *crossing-over*.

Ce type d'algorithme ne permet pas d'obtenir à coup sûr la solution optimale, mais offre un moyen assez rapide d'avoir une bonne idée de cette solution.

4.6.2 Représentation des données

Pour ce genre d'algorithme, on représente toutes les données par une chaîne de bits. Cela peut nécessiter de coder les caractéristiques des individus. Par exemple, si une propriété est à valeur réelle entre 0 et 100, et que l'on désire la coder sur un octet (8 bits), il faudra d'abord la ramener à un nombre compris entre 0 et 255 ($2^8 - 1$). Ce qui peut être fait par une règle de trois : $x' = x/100 * 255$. Ensuite, pour convertir le nombre obtenu, on peut procéder ainsi :

```
for (i = 0 ; i < 8 ; i++)
        {
        c[7-i] = 48 + x' % 2;
        x' /= 2;
        }
```

Si l'on a plusieurs données, on les mettra bout-à-bout dans une même chaîne de bits. Par la suite, on ne manipulera plus que cette chaîne (sauf pour la fonction d'adaptation et pour donner le résultat).

Remarque : en général, on n'aura besoin des fonctions inverses, permettant de passer de la chaîne de bits aux données.

4.6.3 L'algorithme proprement dit

Initialisation

On commence par créer une population aléatoire d'un certain nombre d'individu (50 par exemple). Il ne faut pas avoir un nombre trop faible, mais un nombre très grand n'est pas non plus justifié.

La reproduction

La reproduction dépend de l'adaptation des individus à leur monde. Il s'agit en fait de la fonction que l'on cherche à maximiser. Plus un individu est adapté

(il fournit un résultat élevé à la fonction que l'on veut maximiser), plus il aura de chance de se reproduire.

Pour implanter cette fonction, si la fonction d'évaluation fournit toujours un résultat positif ou nul, on peut procéder ainsi :

```
#include<stdlib.h>
#define TAILLE_POPULATION 50
typedef char *individu;

reproduit(individu *popul_ancienne, *popul_nouvelle);
{
        float adapation[TAILLE_POPULATION];
        float total = 0, tot;
        int i, j;

        for (i = 0 ; i < TAILLE_POPULATION ; i++)
                total += (adaptation[i] = evaluation(popul[i]));

        for( i = 0 ; i < TAILLE_POPULATION ; i++)
                {
                val = total * rand() / (float) RAND_MAX;
                j = 0;tot = 0;

                while (tot+=adaptation[j++] < val);
                copie_individu(popul_ancienne[j-1], &popul_nouvelle[i]);
                }
}
```

Ce procéder permet d'avoir une reproduction relativement proportionnelle (au hasard près) à l'adaptation. Mais une telle répartition n'est pas obligatoire.

La mutation

Une fois que la nouvelle population s'est reproduite, il va falloir la faire muter. Cela se fait en transformant aléatoirement certains bits. Pour de bons résultats, les mutations doivent être assez rare (probabilité entre 10^{-2} et 10^{-4} par exemple).

Cela se fait assez simplement :

```
#define PROB_MUT 1E-4
mute(individu popul[])
{
        int i, j;
```

```
        for (i = 0 ; i <{TAILLE_POPULATION ; i++)
                j = 0 ;
                while (popul[i][j])
                        {
                        if (rand() / (float) RAND_MAX < PROB_MUT)
                                popul[i][j] = 97 - popul[i][j];
                        j++;
                        }
                }
}
```

Le crossing-over

Le crossing over consiste à sélectionner deux individus, et à mélanger intervertir la fin de leur chaîne. Pour de bons résultat, la probabilité de crossing-over doit être assez importante.

```
#define NB_BITS 20
#define PROB_CROSS 0.5
crossing(individu *ancienne_pop, *nouvelle_pop)
{
      int marqueur[TAILLE_POPULATION];
      int i, j;
      int p1, p2;

      for (i = 0 ; i < TAILLE_POPULATION; marqueur[i++]=0);
      for (i = 0 ; i < TAILLE_POPULATION/2;i++)
            {
            while (marqueur[p1 = (int) (rand() / (float) RAND_MAX * TAILLE_POPULATION)]);
            marqueur[p1] = 1;
            while (marqueur[p2 = (int) (rand() / (float) RAND_MAX * TAILLE_POPULATION)]);
            marqueur[p2] = 1;

            if (rand() < RAND_MAX * PROB_CROSS)
                  {
                  position = rand() / (float) RAND_MAX * (NB_BITS-1);
                  for (j = 0 ; j < position ; j++)
                        {
                        nouvelle_pop[i][j] = ancienne_pop[p1][j];
                        nouvelle_pop[i+TAILLE_POPULATION/2][j] = ancienne_pop[p2][j];
                        }
                  for (j = postion ; j < NB_BITS ; j++)
                        {
                        nouvelle_pop[i][j] = ancienne_pop[p2][j];
```

```
                    nouvelle_pop[i+TAILLE_POPULATION/2][j] = ancienne_pop[p1][j];
                    }
                }
            else
                {
                copie_individu(ancienne_pop[p1], &nouvelle_pop[i]);
                copie_individu(ancienne_pop[p2], &nouvelle_pop[i+TAILLE_POPULATION/2]);
                }
            }
}
```

Programme principal

Le programme principal devra se contenter d'appeler successivement les fonctions précédemment données, et cela un certain nombre de fois. Lorsque l'on estime avoir atteint un résultat satisfaisant, il reste à décoder a signification de l'individu le meilleur (ou des meilleurs individus).

4.6.4 Quelques remarques

Ce genre d'algorithme possède de multiples variantes (avec une paire de chromosome, notamment, on peut avoir des allèles récessifs et dominants). Ils permettent d'obtenir très vite de bonnes valeurs. Par contre, on n'a aucune garantie sur le résultat exact.

4.7 Procédures par séparation et évaluation

4.7.1 Introduction

Les problèmes que l'on cherche à résoudre par ce genre de méthode sont des problèmes d'optimisation d'une fonction, avec contraintes de type inégalité sur les variables. De plus, les variables en question ont un nombre déterminé et fini de valeurs possibles. Une sous partie de ces problèmes sont les problèmes à variables binaires, tels que le problème du sac-à-dos. Dans l'exemple qui suit, pour passer à un problème avec plus de valeurs, il suffirait de multiplier le nombre d'embranchements aux nœuds de l'arbre.

4.7.2 Exemple : le problème du sac-à-dos

On désire partir en randonnée. Pour cela, on dispose d'un sac-à-dos de capacité maximum 20l. De plus, on estime ne pas pouvoir porter plus de 12kg. Cependant, on aimerait pouvoir emmener avec soi la liste suivante :

objet	*poids*	*volume*	*interet*
Tente	3	3	7
Rechaud	2	1	6
Eau	6	6	10
Vetementsderechange	2	5	4
Nourriture	2	3	8
$K-Way$	0,5	0,5	5
Duvet	0,5	3	9
Pull	0,5	1	5

Ce problème se traduit, mathématiquement, ainsi on cherche à maxmiser la fonction :

$$F_{max} = 7x_1 + 6x_2 + 10x_3 + 4x_4 + 8x_5 + 5x_6 + 9x_7 + 5x_8$$

avec les contraintes suivantes :

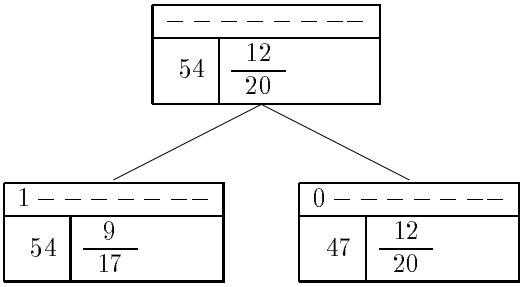
$$\begin{cases} x_i \in \{0,1\} \forall i \in [1,8] \\ 3x_1 + 2x_2 + 6x_3 + 2x_4 + 2x_5 + 0,5x_6 + 0,5x_7 + 0,5x_8 \leq 12 \\ 3x_1 + 1x_2 + 6x_3 + 5x_4 + 3x_5 + 0,5x_6 + 3x_7 + 1x_8 \leq 20 \end{cases}$$

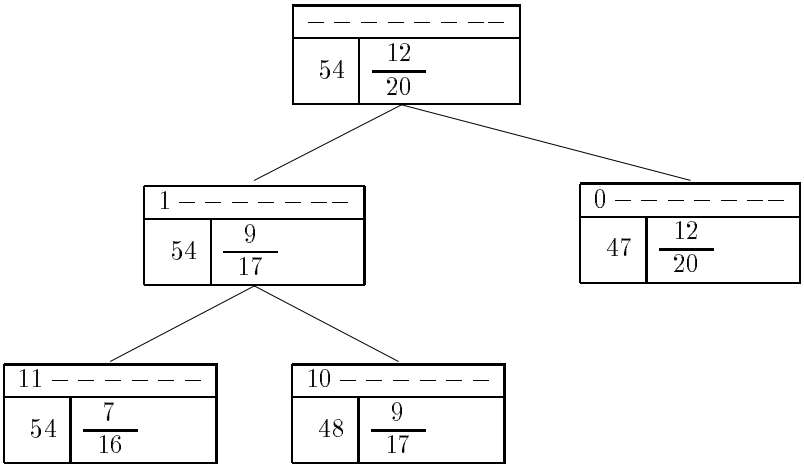
On calcule S, la somme des critères de satisfiabilité (maximum théorique que peut atteindre F_{max}). Dans le cas général (hors du cas binaire), il faut prendre les critères de satisfiabilité multipliés par la valeur maximum que peut prendre la variable associée. Ici, $S = 54$.

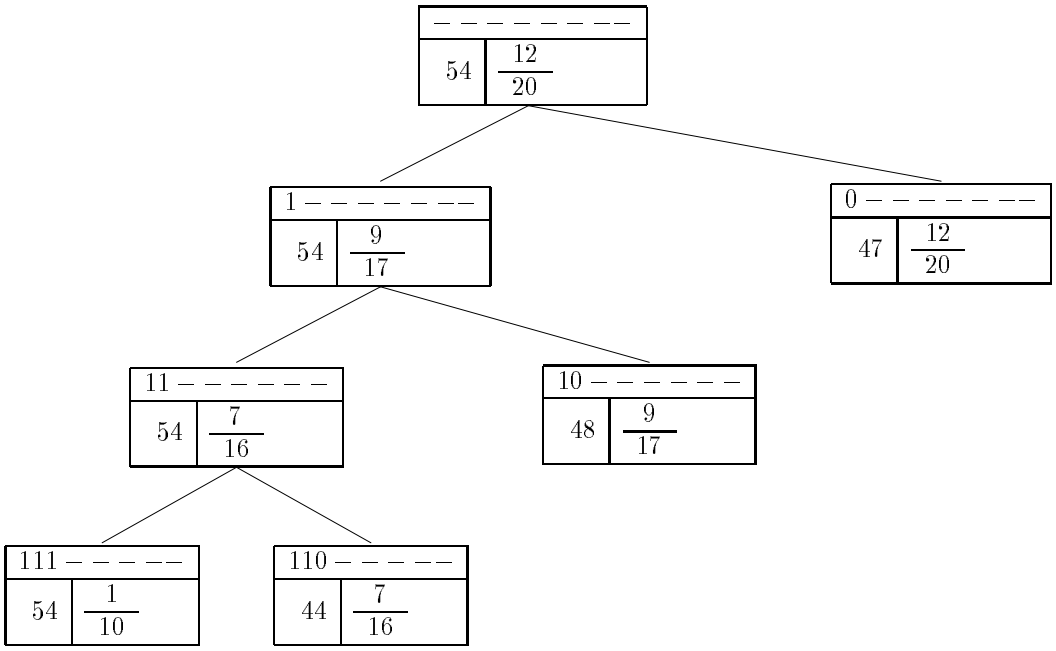
On procède alors selon une structure arborescente : à gauche, on prend l'objet : les poids et volumes restants pour le sac à dos diminuent donc. À droite, on laisse l'objet : c'est donc la satisfaction globale qui diminue.

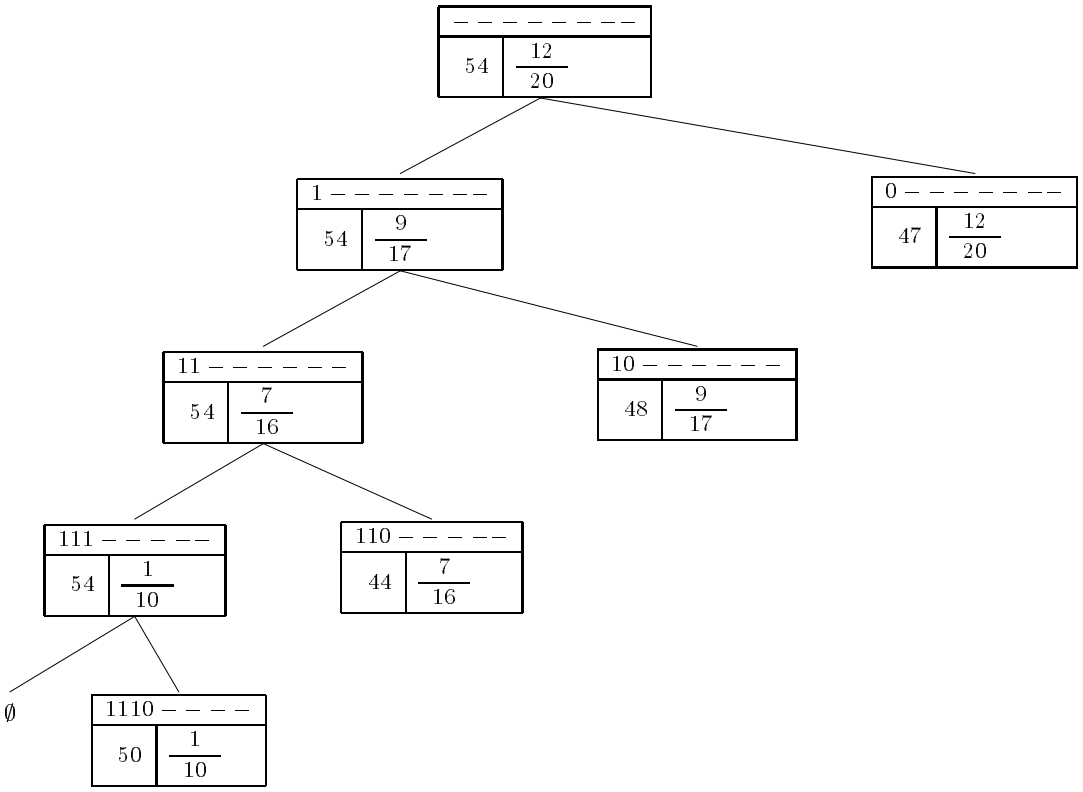
A chaque étape, on prend la feuille ne l'arbre avec le critère S de satisfiabilité le plus grand, et l'on calcule ses fils. Si on obtient des valeurs négatives pour le

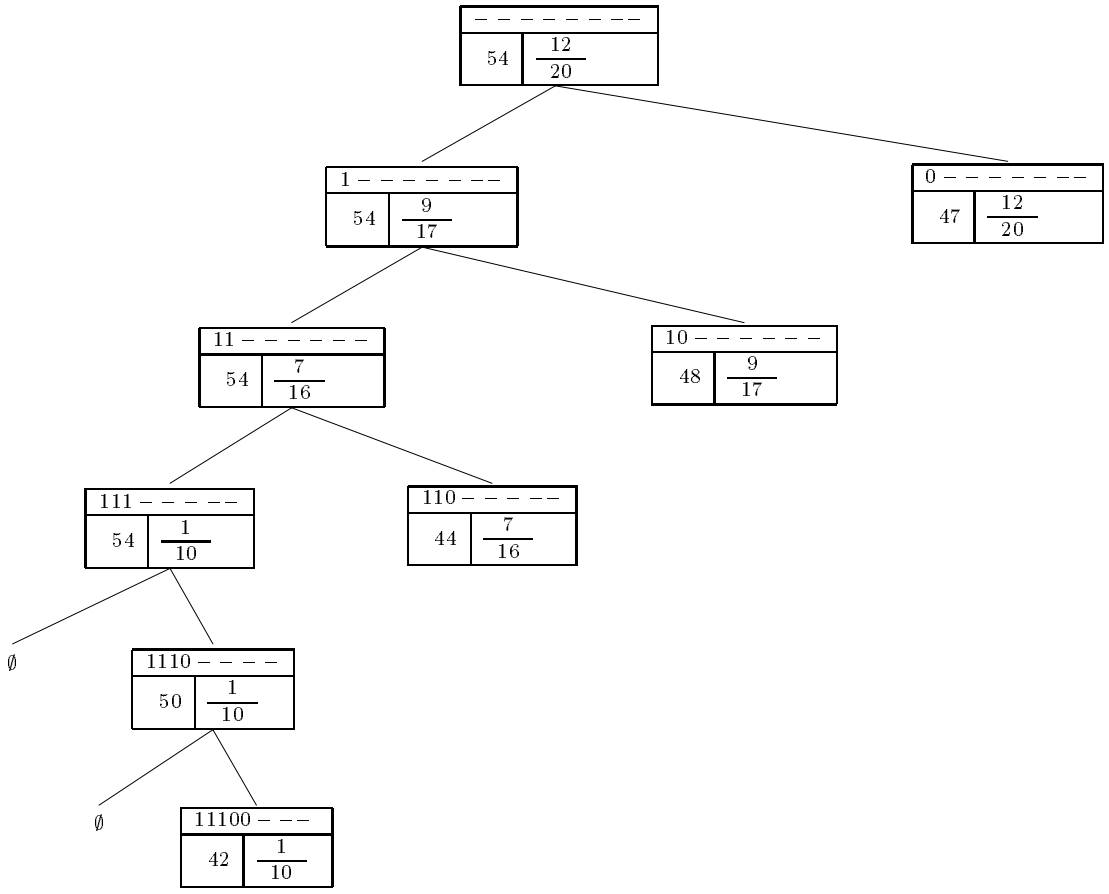
poids ou le volume restant, on considère le fils comme non valide. Si l'une de ces deux contraintes est nulle, et qu'aucun objet n'a de poids ou de volume nul, alors on a obtenu une solution donnant à F_{max} sa valeur maximum.

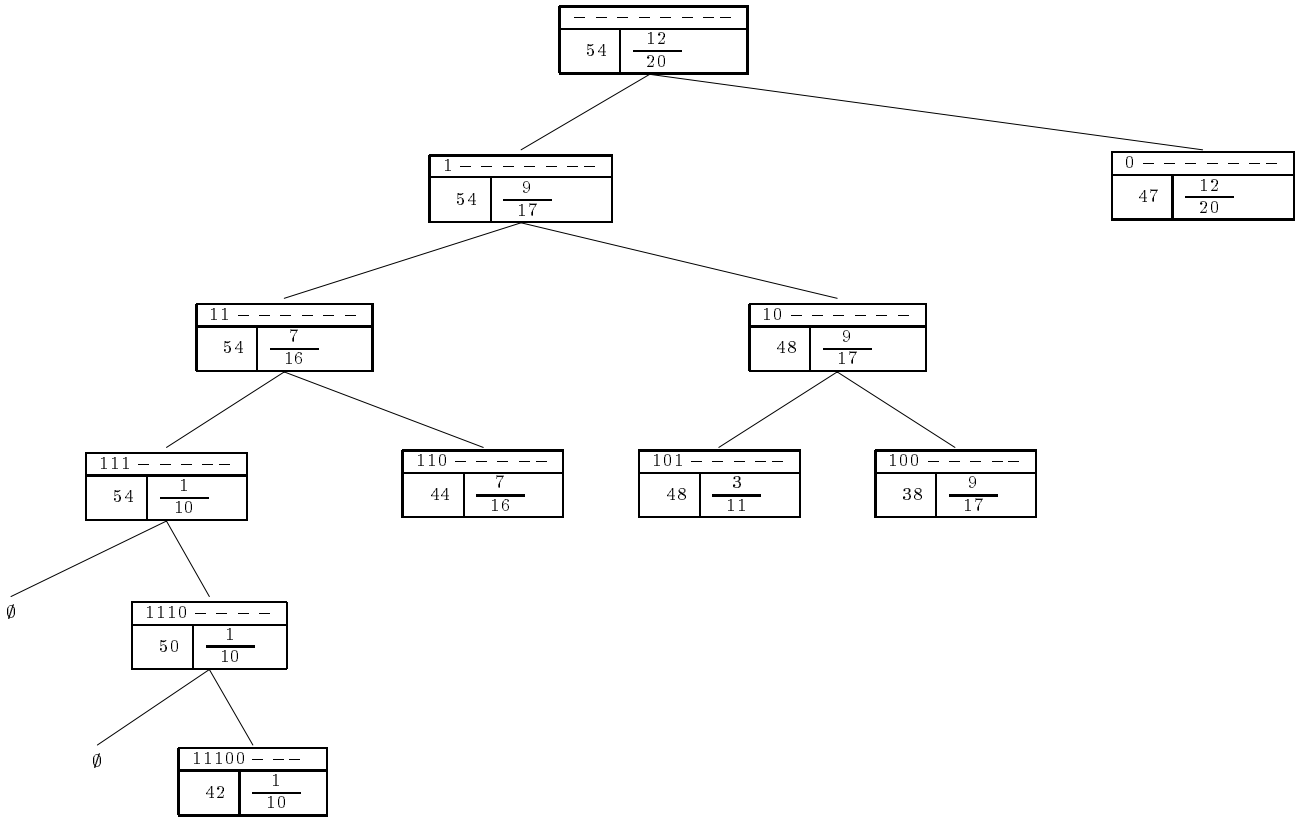


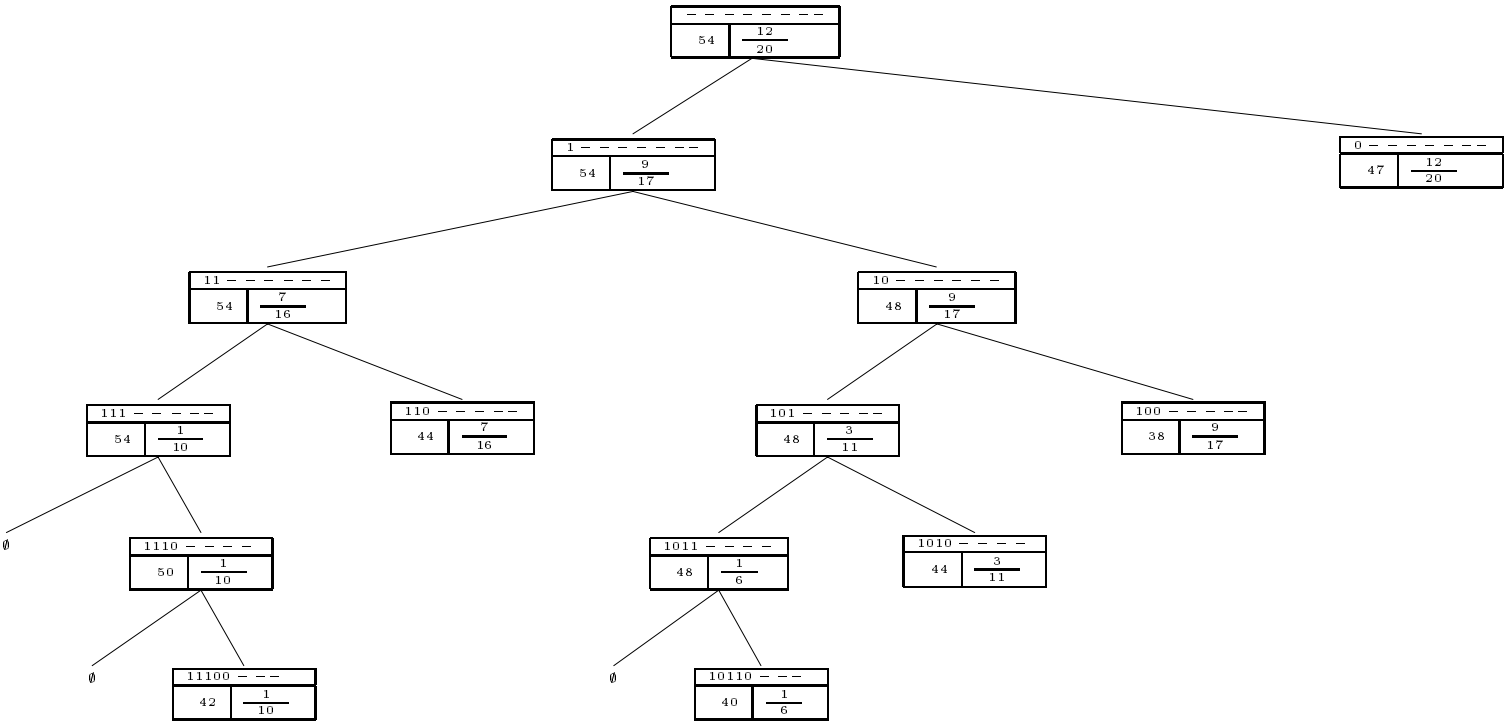


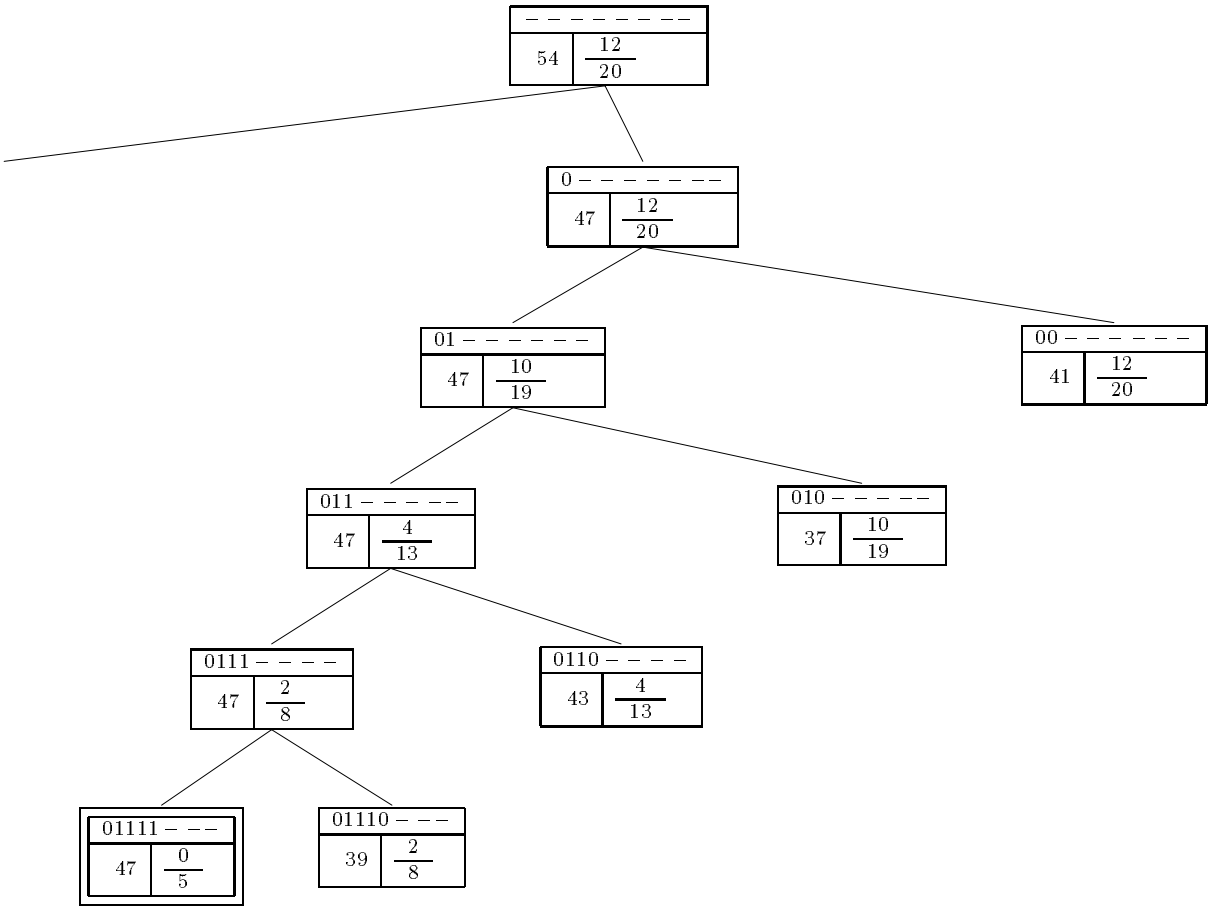

– – – – – – – ––
54
12
20
1 – – – – – – ––
54
9
17
0 – – – – – – ––
47
12
20
11 – – – – – –
54
7
16
10 – – – – – –
48
9
17
111 – – – ––
54
1
10
110 – – – ––
44
7
16


– – – – – – – – –
54
12
20
1 – – – – – – – –
54
9
17
0 – – – – – – – –
47
12
20
11 – – – – – – –
54
7
16
10 – – – – – – –
48
9
17
111 – – – – – –
54
1
10
110 – – – – – –
44
7
16
∅
1110 – – – – –
50
1
10


– – – – – – – –
54 12/20
1 – – – – – – –
54 9/17
0 – – – – – – –
47 12/20
11 – – – – – –
54 7/16
10 – – – – – –
48 9/17
111 – – – – –
54 1/10
110 – – – –
44 7/16
∅
1110 – – – –
50 1/10
∅
11100 – – –
42 1/10


– – – – – – – –
54 12/20
1 – – – – – – –
54 9/17
0 – – – – – – –
47 12/20
11 – – – – – –
54 7/16
10 – – – – – –
48 9/17
111 – – – – –
54 1/10
110 – – – –
44 7/16
101 – – – –
48 3/11
100 – – – –
38 9/17
∅
1110 – – – –
50 1/10
∅
11100 – –
42 1/10


– – – – – – – –
54 12/20
1 – – – – – – –
54 9/17
0 – – – – – – –
47 12/20
11 – – – – – –
54 7/16
10 – – – – – –
48 9/17
111 – – – – –
54 1/10
110 – – – – –
44 7/16
101 – – – – –
48 3/11
100 – – – – –
38 9/17
∅
1110 – – – –
50 1/10
1011 – – – –
48 1/6
1010 – – – –
44 3/11
∅
11100 – – –
42 1/10
∅
10110 – – –
40 1/6



On est tombé sur une solution maximum telle qu'on ne peut plus rajouter d'objet sans dépasser la limite des 12kg. On partira donc avec le réchaud, l'eau, les vêtements de rechange et la nourriture. On peut alors s'interroger sur les critères de satisfiabilité accordés aux différents objets, parce qu'en partant sans tente, pull, et duvet, les nuits risquent d'être dures !

4.7.3 S'aider d'heuristiques

Pour s'arrêter plus tôt, on peut utiliser des heuristiques permettant d'obtenir aisément une solution réalisable. Une première heuristique consiste à prendre les variables par ordre de satisfaction décroissante, et à s'arrêter dès que l'on ne peut plus satisfaire les contraintes. Ici, par exemple, on peut prendre L'eau (10), le duvet (9), la nourriture (8), la tente (7) et le K-way. On a alors une satisfaction de 39. Cela permet de se débarasser tout de suite dans l'arbre des branches avec un résultat inférieur ou égal.

4.7.4 Exercice

On dispose d'un Compact disque que l'on désire recopier sur une cassette de 90 minutes, de telle sorte qu'il reste le moins possible de temps à la fin de la première face. On dispose des chansons suivantes :

1. Le rêve de pêcheur 5'11
2. Liebe 5'07
3. Les nuits sans Kim Wilde 6'15
4. Le pouvoir des fleurs 4'15
5. Bopper en larmes 4'22
6. Le soleil donne 9'20
7. Rockollection 18'19
8. My song of you 4'41
9. Belle île en mer 4'00
10. Du temps qui passe 4'12

Donner une formulation mathématique du problème.

4.8 Programmation dynamique

La programmation dynamique repose sur le *principe d'optimalité* : s'il existe un chemin optimum $C_0, C_1, \ldots C_n$ pour passer de C_0 à C_n, alors pour tout i et j tels que $0 \leq i < j \leq n$, $C_i, \ldots C_j$ est un chemin optimum de C_i à C_j.

4.8.1 Application aux problèmes de programmation linéaire en nombres entiers

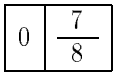
Soit à maximiser la fonction :

$$F_{max} = 4x_1 + 3x_2 + 7x_3 + 5x_4$$

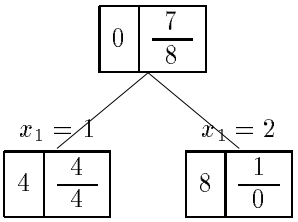
avec les contraintes suivantes :

$$\begin{cases} 3x_1 + 3x_2 + 4x_3 + 3x_4 \leq 7 \\ 4x_1 + 5x_2 + 3x_3 + 2x_4 \leq 8 \end{cases}$$

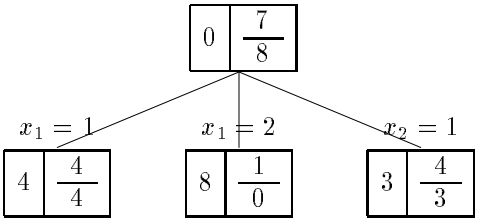
On part de l'état suivant : $F_{max} = 0$, $x_1, \ldots, x_n$ non étudiés :



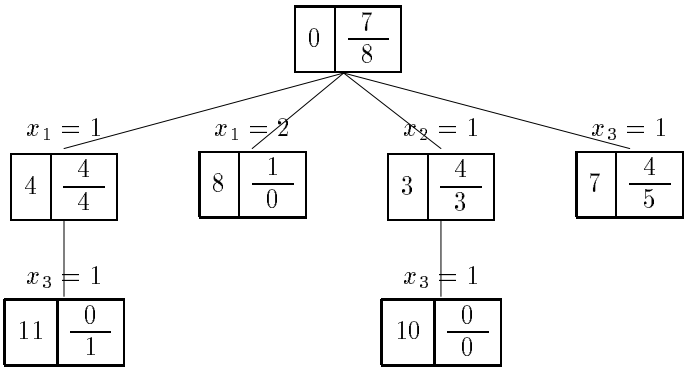
On envisage les valeurs possibles pour x_1 : 0, 1 ou 2 :



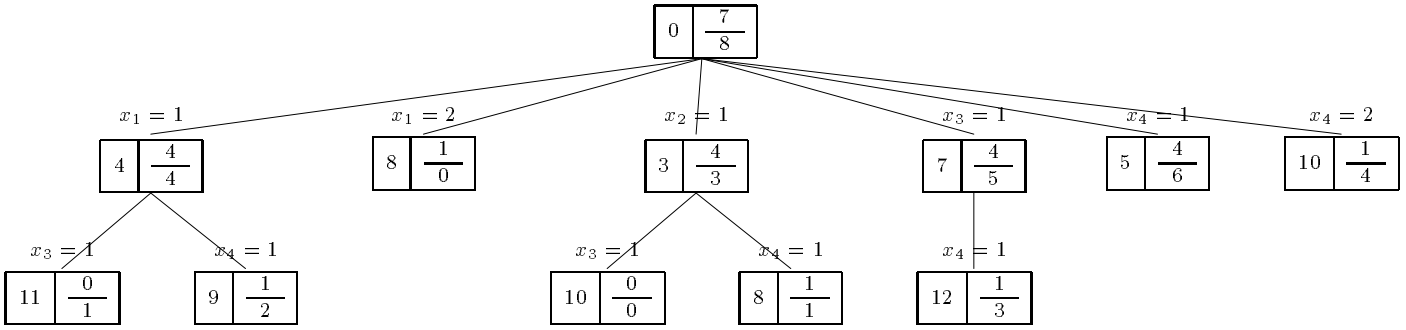
Puis les valeurs possibles pour x_2 : 0 ou 1 :



On passe à x_3, qui peut valoir 0 ou 1 :



Il nous reste enfin à étudier x_4, qui peut prendre les valeurs 0, 1 ou 2 :



On recherche alors le maximum obtenu : on obtient 12. En remontant, on trouve qu'il est atteint pour :

$$\begin{cases} x_4 = 1 \\ x_3 = 1 \\ x_2 = 0 \\ x_1 = 0 \end{cases}$$

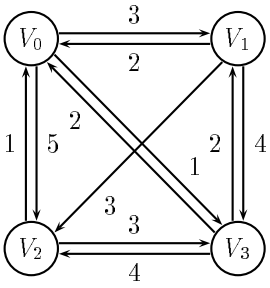
4.8.2 Application au problème du voyageur de commerce

Énoncé

Le problème est le suivant : un voyageur de commerce part d'une ville V_0. Il doit visiter n villes, reliées entre elles directement ou indirectement. Si deux villes V_i et V_j sont reliées directement par une distance d, on note $D_{i,j} = d$ la distance qui les sépare. Sinon, on pose $D_{i,j} = \perp$.

Le souhait du voyageur de commerce est bien évidemment de visiter toutes les villes, mais en parcourant une distance minimale. De plus, il peut désirer revenir à son point de départ. Enfin, il ne désire pas repasser deux fois par la même ville.

Exemple



4.9 Calcul scientifique

4.9.1 Approximation au sens des moindres carrés

Présentation

Il s'agit d'approximer une courbe donnée par un ensemble de points par une composition linéaire d'autres fonctions, en minimisant la distance des points à la courbe résultat. Souvent, on prend pour fonctions de base les polynômes x^i.

Principe

Soit f la fonction résultat, $B = \{B_0, B_1, \ldots B_n\}$ les polynômes de la base, et $P = \{(x_i, y_i), i \in [0, p]\}$ l'ensemble des points dont on dispose. On a de plus $p > n$.

f a donc la forme suivante :

$$f(x) = \sum_{i=0}^{n} a_i . B_i(x)$$

où, par définition, on a : $\sum_{i=0}^{p}(y_i - f(x_i))$ minimum.

Cela peut être obtenu en résolvant le système linéaire suivant :

$$V.a = S$$

avec :

$$\begin{cases} V = [v_{i,j}], \text{ avec } v_{i,j} = \sum_{k=0}^{p} B_i(x_k).B_j(x_k) \\ a = [a_i], i \in [0, n] \\ S = [s_i], \text{ avec } s_i = \sum_{j=0}^{p} B_i(x_j).y_j \end{cases}$$

4.9.2 Résolution de systèmes linéaires par itération

Principe général

Les méthodes type pivot de Gauss sont souvent lourdes à mettre en œuvre. De plus, la présence de grand ou petits pivots peut entraîner de lourdes erreurs. Les algorithmes présentés ici obéissent à un autre principe : on ne cherche pas une solution exacte, mais une solution approchée. De plus, le principe itératif utilisé est très simple.

Méthode de Jacobi

On commence par réorganiser le système linéaire de façon à ce que chaque équation exprime la valeur d'une inconnue en fonction des autres. Soit, par exemple, le système suivant :

$$\begin{cases} 5x_0 + x_1 + x_2 = 8 \\ 2x_0 + 4x_1 + x_2 = -7 \\ -x_0 + 2x_1 + 3x_2 = 12 \end{cases}$$

On le réécrit ainsi :

$$\begin{cases} x_0 = \frac{1}{5}(8 - x_1 - x_2 \\ x_1 = \frac{1}{4}(-7 - 2x_0 - 4x_2) \\ x_2 = \frac{1}{3}(12 + x_0 - 2x_1) \end{cases}$$

Puis, en partant de valeurs initiales, on calcule le vecteur $x = [x_0, x_1, x_2]$ à l'étape n en fonction des valeurs de x à l'étape $n-1$. Si l'on converge, on obtient une solution du système. Dans le cas d'un système admettant une infinité de solution, on n'en trouvera qu'une, qui dépendra du vecteur initial.

À noter : une condition suffisante de convergence est que la matrice de départ du système soit *à diagonale dominante* :

$$\forall i : \left(\sum_{j=1}^{n} (1 - \delta_{i,j}) \, |a_{i,j}| \right) < |a_{i,i}|$$

avec $\delta_{i,j}$ symbole de Kronecker : vaut 1 si $i = j$, 0 sinon.

Méthode de Gauss-Seidel

L'algorithme précédent nécessite, lorsque l'on calcule le vecteur x à l'instant t, d'avoir en mémoire le vecteur x à l'instant $t-1$ tout au long du calcul. Il faut donc avoir deux versions de x en mémoire. La méthode de Gauss-Seidel évite ce problème : pour calculer la i-ème composante à l'étape t, on utilise les coordonnées $x_j, j < i$ à l'étape t, et les coordonnées $x_j, j > i$ à l'étape $t-1$.

Méthode de relaxation

Il s'agit d'une extension pondérée de la méthode de Gauss-Seidel : la partie droite du calcul de x_i à l'étape t fait intervenir x_i à l'étape $t-1$ avec un facteur $1-\omega$, et le reste (identique à la méthode de Gauss-Seidel) avec un coefficient ω. On constate alors que la méthode de Gauss-Seidel correspond au cas particulier $\omega = 1$.

On notera qu'un critère nécessaire de convergence est d'avoir $0 < \omega < 2$.

4.9.3 Résolution d'une équation non linéaire

Théorème du point fixe

Toute application contractante f d'un espace métrique complet E dans lui-même admet un point fixe unique x^* et pour tout x de E, la suite $(x)_t$ définie par $x_t = f(x_{t-1}$ converge vers x^*.

Application à la résolution d'équations $x = f(x)$

En général, il existe un intervalle inconnu autour de la racine sur lequel la fonction f est contractante. On peut donc appliquer l'algorithme.

Celui-ci consiste à appliquer d jusqu'à avoir une erreur absolue ($|x_t - x_{t-1}|$) ou une erreur relative ($|\frac{x_t - x_{t-1}}{x_t}|$) inférieure à un seuil fixé. Par mesure de précaution (en cas de divergence), on peut aussi contrôler le nombre d'itérations effectuées.

Enfin, pour vérifier qu'il n'y a pas eu d'erreur due à des problèmes d'arrondi, il faut vérifier que la fonction $x - f(x)$ change de signe sur $[x_t - \epsilon; x_t + epsilon]$.

Technique de sur-itération

Pour accélérer la convergence (ou obtenir une convergence là où la méthode précédente échoue), on peut utiliser une technique de sur-itération : cela consiste à réécrire l'équation $x = f(x)$ sous la forme $x + \gamma.x = f(x) + \gamma.x$, ce qui donne $x = g(x) = \frac{f(x)+\gamma.x}{1+\gamma}$, avec $g'(x^*) = 0$, soit $\gamma = -f'(x^*)$

Comme la valeur de x^* n'est pas connue, c'est encore plus le cas de celle de $f'(x^*)$. On utilise donc pour approximer $x*$ la valeur courante du calcul, à savoir x_t.

Équations du type $f(x) = 0$: Newton-Raphson

Sous certaines conditions, en développant $f(x)$, on a qu'il existe un ξ tel que :

$$f(x^*) = f(x_t) + (x^* - x_t)f'(x_t) + \frac{1}{2}(x^* - x_t)^2 f''(\xi)$$

Or pas hypothèse, on a $f(x^*) = 0$. D'où :

$$x^* = x_t - \frac{f(x_t)}{f'(x_t)} - \frac{(x^* - x_t)^2}{2f'(x_t)} f''(\xi)$$

Ce qui, en négligeant les termes d'ordre 2, donne :

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$$

4.9.4 Résolution d'un système d'équations non-linéaires f(x)=0

La technique précédente peut être utiliser en passant par les dérivées partielles. Ainsi, si l'on a un système de la forme :

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \ldots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases}$$

On peut le réécrire sous la forme:

$$\begin{cases} x_{1,t+1} = x_{1,t} - \frac{f_1(x_{1,t},x_{2,t},\ldots,x_{n,t})}{\frac{\partial f_1}{\partial x_1}(x_{1,t},x_{2,t},\ldots,x_{n,t})} \\ x_{2,t+1} = x_{2,t} - \frac{f_2(x_{1,t+1},x_{2,t},\ldots,x_{n,t})}{\frac{\partial f_2}{\partial x_2}(x_{1,t+1},x_{2,t},\ldots,x_{n,t})} \\ \ldots \\ x_{n,t+1} = x_{n,t} - \frac{f_n(x_{1,t+1},x_{2,t+1},\ldots,x_{n-1,t+1},x_{n,t})}{\frac{\partial f_n}{\partial x_n}(x_{1,t+1},x_{2,t+1},\ldots,x_{n-1,t+1},x_{n,t})} \end{cases}$$