Sommaire

Chapitre 1

Introduction

1.1 But du cours

Apprendre à se créer des structures de données que l'on **utilisera** quand on **programmera plus tard des applications**.

1.2 Comment ferons-nous ?

1.2.1 Définition de concepts

On se définira des structures de données, d'abord simples, puis de plus en plus compliquées, et cela indépendament de toute implantation sur machine. En effet nous parlons tout à fait naturellement de concepts tels que ensemble, suite, espaces vectoriels etc... sans même avoir besoin de les décrire explicitement... on sait ce que ça veut dire.

1.2.2 Spécification axiomatiques

On spécifira exactement ces notions par ce qu'on appelle des spécifications algébriques de types abstraits (ref. J.F.Dufourd TSI)

1.2.3 Programmation

Ces structures de données ainsi spécifiées sont **implantées** par un programme.

1.2.4 Bibliothèque

Il est trés important de se construire ces programmes une fois pour toute et de les ranger dans une bibliothèque afin de ne plus y revenir.

1.3 Couches

Nous crérons nos stuctures de données par **couches** successives.
D'abord des choses assez simples telles que ensembles, files, piles, puis des structures plus compliquées telles que graphes, arbres, etc...
Il est **trés important de ne pas mélanger** les différentes couches et de ne faire référence aà;une couche donnée qu'à la couche elle-même et à la couche immédiatement inférieure.Ainsi les ensembles peuvent faire réference au pointeurs, les graphes au ensembles mais jamais aux pointeurs.

1.3.1 Ce qu'on utilise partout

A tout niveau peut être fait référence aux types:

- entier
- flottant
- logique

- stucture en temps qu'entité entière si elle est définie à un niveau inférieur
- détail d'une structure si elle est définie à ce niveau là

Ainsi le nombre de sommets d'un graphe défini par une structure sera noté *NbrSommets(G)* pour les fonctions de niveau supérieur où ce graphe est défini plustôt que *G.NbrS* qui est en fait la vraie définition.
C'est une société de **classes**... en quelque "sorte".

1.4 Vol au dessus de ce qu'on va faire

1.4.1 Qu'utilise-t-on sans réfléchir?

- les scalaires... entiers, flottants, logiques
- les structures simples
  - { entier; entier }
  - { entier; flottant }
  - { flottant; entier; flottant }
  - ... bref des n-uplets avec mélange de type.
- les structures imbriquées sont à priori à éviter, ce serait du mélange de couches, il vaut mieux les définir correctement avec les opérateurs d'accés et donc de mettre ces nouveaux types (sortes) dans la structure.
- les tableaux, uniquement quand ils sont conformes à ce qu'on veut, ainsi une matrice de rotation sera bien sûr une vraie matrice 3x3, un vecteur dans R^3 un tableau de 3 réels...

  Il faut éviter de considérer un tableau de noms comme un ensemble de noms; comment fera-t-on une adjonction, une suppression? Un tableau n'est souvent qu'un outil, une structure élémentaire.

1.4.2 Au programme

- les scalaires
- les n-uplets
- les ensembles
- les files,piles
- les listes (triees, tampon circulaire ,etc...)
- les graphes (simples, valués, avec arcs uniques ou multiples, arcs valués, etc...)

- les arbres
- quelque chose de plus gros.

et bien sûr nous programmerons de grandes choses avec tous ces beaux outils.

Chapitre 2

Ensembles

2.1 Spécifications

Les opérations élémentaires nécessaires sont:

- ensemble inclu(x, e)
- ensemble retire(x, e)
- **booleen** exist(x, e)
- **typeT** nimporte(e)
- **booleen** vide(e)
- ensemble ensemblevide()

que nous définirons par les axiomes suivants:

$$
\begin{array}{ll}
(1) & \text{vide(ensemblevide())} = \textbf{vrai} \\
(2) & \text{vide(inclu}(x, e)) = \textbf{faux} \\
(3) & \text{exist}(x,\text{ensemblevide())} = \textbf{faux} \\
(4) & \text{exist}(x,\text{inclu}(y, e)) = \\
 & \qquad\qquad \textbf{si } (x == y) \textbf{ alors vrai} \\
 & \qquad\qquad\qquad\qquad \textbf{sinon } \text{exist}(x, e)) \\
(5) & \text{retire}(x,\text{ensemblevide())} = \text{ensemblevide()}
\end{array}
$$

(6) retire(x,inclu(y, e)) =
si ($x == y$) **alors** retire(x, e)
sinon inclu(y,retire(x, e))

(7) **si non** vide(e) **alors**
exist(nimporte(e),e) = **vrai**

Nous supposerons que cette spécification est complète et consistante et proposons quelques implantations (voir listings)
Nous pouvons maintenant créer les opérations de la couche immédiatement supérieure à savoir:

- ensemble union($e1, e2$)
- ensemble intersection($e1, e2$)
- ensemble complement(e, E)
- **integer** cardinal(e)

et pourquoi pas:

- ensemble_d_ensembles parties(e)

2.2 Applications immédiates

- compter le nombre d'éléments positifs dans un ensemble d'entiers.
- créer l'ensemble des entiers pairs pris dans e.
- créer les ensembles de "clusters" d'un ensemble de points c'est-à-dire qu'il faut créer tous les ensembles e_i tels que tout point de e_i est situé à une distance inférieure à D de tous les autres éléments de e_i et que tout point de e situé à une distance inférieure à D soit dans e_i.

2.3 Dynamique-Statique

La programmation de tout ça est évidemment simple si on ne tient pas compte de la notion "Dynamique-Statique", il suffit en général de transcrire dans le langage souhaité les spécifications "étudiées pour".
Revenons à nos ensembles, on voit qu'il y a un problème pour *exist()* et *retire()*, l'un se servant de l'autre suivant la manière de programmer et, surtout pour *retire()*, on voit qu'il faut introduire une nouvelle fonction qui doit fournir par le même appel n'importe quel élément de e et l'ensemble obtenu par exclusion de cet élément de e.

2.3.1 Programmons,

booleen exist(x, e)
    si vide(e) **alors vrai**
        sinon y=nimporte(e)
            ($y == x$ ou exist(x,retire(y, e)))

...jusque là pas de problème si ce n'est que *retire()* ne doit pas utiliser *exist()*...et encore c'est pas sûr.

ensemble retire(x, e)
    si vide(e) **alors** ensemblevide()
        sinon (y, r) = exclusion(e)
            Si $x == y$ **alors** r
                sinon inclu(y,retire(x, r))

Il a fallu rajouter la fonction *exclusion(e)* qui fournit deux résultats, un élément et un ensemble.

2.3.2 Exercices

Ensembles des parties de e.

Il faut évidemment créer une structure d'ensemble d'ensembles.

Nous supposerons qu'elle existe. Nous utiliserons les mêmes notations pour Exist(e, ee), Retire(e, ee), Inclu(e, ee), Vide(ee) et Nimporte(ee) en mettant la majuscule. Nous noterons Ensemble_d_ensembles_Vide() la fonction qui crée un ensemble vide d'ensembles.

On écrit alors simplement (mais en réfléchissant beaucoup) le programme suivant:

```
Ensemble_d_ensembles Epe(e)

    ensemble e

    Epe(e) = si Vide(e)
                 alors Inclu(ensemblevide()
                             ,Ensemble_d_ensembles_Vide())
                 sinon
                     x = nimporte(e)
                     r = retire(x, e)
                     Union(Epe(r),inclupartout(y,Epe(r))
```

```
Ensemble_d_ensembles inclupartout(y, Ee)

    float y,Ensemble_d_ensembles Ee

    si Vide(Ee) alors Ensemble_d_ensembles_Vide()
               sinon
                   e = nimporte(Ee)
                   Inclu(inclu(y, e),inclupartout(y,retire(e, Ee)))
```

Chapitre 3

Piles, Files et Listes

3.1 Introduction

Les ensembles forment la structure de données la plus simple, nous allons rajouter une information en donnant un ordre sur les éléments. D'où la notion de piles et de files.

3.2 Piles

3.2.1 Les opérations élémentaires

- sommet(p)
- depile(p)
- empile(x, p)
- pilevide()
- vide(p)

3.3 Files

3.3.1 Les opérations élémentaires

- tete(f)

- corps(f)
- enfile(t, x) (ou adjq(t, x))
- filevide()
- vide(f)

Rien à dire de plus que pour les ensembles, c'est même plus simple. Notons qu'on peut programmer les files à l'aide des piles mais que la réciproque n'est pas vraie.

3.3.2 Le jeu de bataille

Principe

Il s'agit d'écrire un programme qui simule le jeu de cartes de bataille.
Il doit **distribuer** les cartes, puis **afficher** la séquence correspondant à une partie et finalement **déterminer** quel est le gagnant.

Déroulement d'une partie

Chaque joueur a une file de cartes, ils posent chacun la tête de leur file sur la table. Celui qui a la plus grande carte ramasse les cartes, d'abord celle de son adversaire puis la sienne et les met en queue de sa file. Si les deux cartes ont les mêmes valeurs, chacun des joueurs place à l'envers la tête de sa file sur sa pile sur la table puis la tête de sa file sur sa pile. Celui qui a la plus grande carte ramasse comme précédemment les deux piles. Est déclaré perdant celui qui n'a plus de carte dans sa file. Lorsque les deux n'ont plus de carte il y a égalité.

Programmons

si lui **et** moi **alors** egalite
si mafile **est vide alors** lui
si safile **est vide alors** moi

jegagne = macarte > sacarte
ilgagne = macarte < sacarte

mafile = **si** jegagne **alors**
mafile = enfile(mafile,sapile)
mafile = enfile(mafile,mapile)
safile = **si** ilgagne **alors**
safile = enfile(safile,mapile)
safile = enfile(safile,sapile)
mapile = empile(macarte,mapile)
sapile = empile(sacarte,sapile)

macarte = tete(mafile)
sacarte = tete(safile)

mafile = corps(mafile)
safile = corps(safile)

etc..

3.4 Listes

Les listes sont construites à partir des files en ne permettant pas l'adjonction en queue mais des opération spécifiques. Citons :

- les listes **triées** et qui le restent après insertion (insav(x, l) ou insap(l, x)).
- les listes **circulaires** que nous appelerons des carroussels (voir plus loin).
- les listes **doublement chaînées**. Notons ici que ce type de liste n'apporte rien de neuf si ce n'est une exécution plus rapide du fait de l'opérateur *précedent()*.
- etc., on peut imaginer tout ce qu'on veut pour améliorer l'exécution de l'une ou l'autre des opérations (adjonction, recherche, suppression, etc.).

3.5 Carroussel

3.5.1 Les opérations élémentaires

sorte carroussel;

utilise T;

pour tout x **de type** T **et tout** carroussel c
on dfinit:

carroussel carrousselvide()
carroussel rotation(c)
carroussel inclusion(x, c)
carroussel extraction(c)
T element(c)

3.5.2 Flavius

Avec cela il est facile de programmer Flavius ...

> *"... les méchants romains voulaient tuer 39 des 40 esclaves qu'ils avaient capturés. Ils les font mettre en cercle, comptent 1, 2, 3, 4, 5, 6 et tuent le septieme, puis comptent etc.. Où le futé Flavius, esclave parmi les 40, devait-il se mettre?[1]"*

int Flavius()

{
carroussel c;

c=carrousselvide();

[1] voir sujet d'examen J.F.Dufourd ou DNA janvier 1989

```
for (i = 1; i <= 40; i++) c=inclusion(i, c);

for (i = 1; i <= 39, i++)
        {
        for (k = 1; k <= 7; k++) rotation(c);
        extraction(c);
        }
imprimer(”la bonne place est :”,element(c));
}
```

Voici à titre éducatif le programme C récursif qui tourne sans structure de données particuliaire:

```
main()

     {
     printf(”le bon numero est :%d\ n”, bonnumero(0,40)+1);
     }



int bonnumero(i, n) /* on a n elements ,
                       on vient d’eliminer
                       celui qui est situe
                       avant le numero i */

       int i, n ;

       {
       int k,v;

            if (n == 1) return(0);
```

v=$((i-1)+7)\%n$;
k=bonnumero(v, $n-1$);

if $(k < v)$ **return**(k);
if $(k >= v)$ **return**$(k+1)$;
}

3.5.3 Serveur d'imprimantes

Un serveur d'imprimante est un programme qui tourne sur un ordinateur sur lequel sont connectés:

- un réseau par où arrivent les informations à imprimer on supposera que l'on reçoit des paquets de la forme:

  o i t c

  où

  - **o** est un numéro d'ordinateur
  - **i** est un numéro d'imprimante
  - **t** est 0 ou 1 suivant que le caractère c est une commande ou du texte.
  - **c** est le caractère ou la commande

- un ensemble d'imprimantes sur des ports RS232 normaux. (l'imprimante i sera connectée au port i)

Sur le réseau peuvent être connectés autant d'ordinateurs qu'on veut dans la mesure où ils ont tous un numéro différents.

On suppose aussi pour simplifier que tout se passe sans erreurs.

D'après le protocole suivant:

repeter indefiniment

- **lecture** eventuelle d'un paquet en reception
(si yaPAQUET (variable systeme) est vrai)

- **emission** d'un caractere sur une imprimante i
(si impLIBRE(i) est vrai
(variable systeme associee au port i))

ep **fin r**

Notons bien qu'on ne peut pas émettre plus d'un caractère sur une seule imprimante entre deux lectures sur le réseau.

L'impression d'un message par un ordinateur consiste à envoyer:

- la commande **sot** (**S**tart **O**f **T**exte)

- les caractères composant le texte un par un

- la commande **eot** (**E**nd **O**f **T**exte)

Un même ordinateur peut envoyer simutanément des fichiers vers des imprimantes différentes.

Plusieurs ordinateurs peuvent envoyer simultanément des fichiers vers la même imprimante.

Stuctures de données nécessaires:

- Il nous faut un **ensemble de couples o**rdinateurs/**i**mprimantes. A chaque couple **(o,i)** on associe une file qui est le message en réception.
  Ce couple **(o,i)** n'existe que pendant **sot** … **eot**.
- Il nous faut une **liste** par imprimante **triée** en fonction de priorités de tailles pour les fichiers en attente d'impression. (on peut éventuellement n'avoir qu'une seule liste pour toutes les imprimantes)
- Il nous faut un **carroussel** en sortie pour émettre vers les imprimantes.

Chapitre 4

Tables

4.1 Avertissement

L'essentiel de ce chapitre réside dans l'**implantation** des structures que l'on définies.

4.2 Introduction

Les ensembles c'est bien. Les ensembles de couples c'est déjà plus délicat, on constate que la notion d'ensembles ne s'applique plus trés bien parce que l'on est souvent amené à rechercher un élément du couple en fonction de l'autre et que ceci ne peut se faire qu'en parcourant bêtement tout l'ensemble jusqu'à trouver le bon couple.
On aimerait bien accélérer cette recherche.

Et Dieu créa les tables.

Nous définirons une tables comme étant un ramassis (pour ne pas dire ensemble) de couple (i, v) où i et v ne jouent pas le même rôle. "i" est l'entrée, "v" la valeur. Les recherches consisteront essentiellement à trouver v en fonction de i.

Trés important: à une entrée i ne correspond qu'une valeur v.

En fait notre parallèle couples/table se résume au fait que l'on peut implanter une table à l'aide d'un ensemble de couples.

4.3 Définition et Spécifications

Soit I un ensembles d'entrées, on veut associer à chaque entrée $i \in I$ une valeur v défnie par la table t. Soit T l'ensemble de valeurs possibles. Nous noterons $\mathcal{T}$ l'ensemble des tables $t : I \to T$.

4.3.1 Définissons les opérations

tabvide	:	$\to \mathcal{T}$
adj	$: \mathcal{T} \times I \times T$	$\to \mathcal{T}$
sup	$: \mathcal{T} \times I$	$\to \mathcal{T}$
mod	$: \mathcal{T} \times I \times T$	$\to \mathcal{T}$
elem	$: \mathcal{T} \times I$	$\to \mathcal{T}$
vide	$: \mathcal{T}$	$\to \{$**vrai** , **faux** $\}$
exist	$: \mathcal{T} \times I$	$\to \{$**vrai** , **faux** $\}$

4.3.2 avec les préconditions

pre(adj(t, i, v)) = **non** exist(t, i)

pre(elem(t, i)) = exist(t, i)

4.3.3 et les spécifications

vide(tabvide())=**vrai**

vide(adj(t, i, v))=**faux**

elem(adj$(t, i, v), j$)=**si** $i = j$ **alors** v **sinon** elem(t, j)

mod(t, i, v))=adj(sup$(t, i), i, v$)

sup(adj$(t, i, v), j$) = **si** $i = j$ **alors** t **sinon** adj(sup$(t, j), i, v$)

4.3.4 Notation

Nous noterons $t(i)$ l'élément elem(t, i).

4.4 Matrices

Dans ce qui suit nous ne parlerons que de matrices à deux dimensions mais tout se généralise à autant de dimensions que l'on veut.

Lorsque l'on programme des fonctions travaillant sur des matrices $m \times n$ on se rend compte qu'il y en a trois types.

a. m et n connus à la compilation
b. m et n inconnus à la compilation
c. les matrices creuses et trop grandes pour tenir en mémoire.

On pourrait évidemment tout programmer en utilisant bêtement les opérations sur les tables définies plus haut,... en ayant choisi une quelconque implantation.
Nous choisirons une **implantation différente** pour chacun des trois cas.

4.4.1 m et n connus à la compilation

On peut utiliser le type matrice du langage c et programmer avec les notations

$$A[i][j]$$

m et n étant des constantes définies par exemple par

$$\#\textbf{define } m \ 18$$
$$\#\textbf{define } n \ 12$$

On peut éventuellement imaginer que m ne soit pas une constante.

4.5 m et n inconnus à la compilation

Il faut gérer soi-même le calcul des adresses et faire l'adressage des éléments à l'aide des pointeurs.

Si A est une matrice $m \times n$ déclarée

float $A[m][n]$;

l'adresse de l'élément $A[i][j]$ est obtenu par $*(A + i * n + j)$.

Petite remarque concernant le calcul:

si on écrit

typedef float ligne[n] /*n constant */
typedef ligne $*A$

on a alors

$$A[i][j] \leftrightarrow *(*(A+i)+j)$$

malheureusement ceci ne marche que si n est connu à la compilation.
Question: comment déclarer A sans utiliser **typedef** ?
Je pense que **float** $(*A)[n]$ devrait marcher.

4.6 Une structure de données *matriceMN*

Pour cela on définit les types:

typedef struct s_matriceMN
{**int** m, n; **float** $*t$} desc_matriceMN, matriceMN;

on crée une matrice par

matriceMN A;

A = **malloc**(**sizeof**(desc_matriceMN));
$A \to m = m$;
$A \to n = n$;
$A \to t =$ **malloc**($m * n *$ **sizeof**(**float**));

et on remplit le tableau $A \to t$.

4.7 Matrices creuses, chaînage orthogonal et *merdique*.

4.7.1 On se situe

Il s'agit de stocker de trés grandes matrices dont la plupart des éléments sont nuls. Toute valeur non stockée est supposée nulle. Nous supposerons également que ces matrices servent essentiellement à faire des produits, à gauche et à droite, donc que la matrice est à considérée comme étant à la fois composée de vecteurs *ligne* et de vecteurs *colonne*.

4.7.2 Quelle implantation

Ce pourquoi nous proposerons une implantation **orthogonale**, c'est-à-dire que chaque vecteur est à considéré comme une liste ordonnée de nombres. Bref, à chaque élément non nul est associé

- le prochain élément non nul dans la ligne
- le prochain élément non nul dans la colonne

On veut également, pouvoir associer à chaque ligne, la prochaine ligne non entièrement nulle, à chaque colonne la prochaine colonne.

4.7.3 Pour quelles opérations

On veut

- première ligne non nulle
- première colonne non nulle
- prochaine ligne non nulle
- prochaine colonne non nulle
- pour une ligne donnée, premier élément non nul
- pour une colonne donnée, premier élément non nul
- pour un élément donné, prochain élément non nul dans la ligne

- pour un élément donné, prochain élément non nul dans la colonne

En fait, je pense que l'on extraira essentiellement des vecteurs pour les injecter dans des fonctions du style produit scalaire etc..
Il faut évidement pouvoir modifier les valeurs de la matrice. Attention, en cours de programme, il faut toujours savoir dans quelle ligne et colonne on se trouve, il faut stocker cette information pour chaque élément.

4.8 Table d'identificateurs

Ceci est un exemple de table.
Au cours d'une compilation, le compilateur rencontre des identificateurs de variables. A chaque variable est associé un type, par exemple **int** , **float** , etc.. Ce type est déterminé par l'instruction de déclaration de la variable, qui devrait généralement apparaître avant toute autre utilisation. Certains compilateurs admettent néanmoins que les variables soient citées avant d'être déclarées; nous leur associerons le type **inconnu** (ou "non encore défini"). Une variable apparaît généralement un grand nombre de fois au cours d'une compilation, et à chaque fois, il est nécessaire de vérifier son type. Cette recherche doit donc être trés rapide. Un programme peut contenir des centaines de variables, et même plus.
Différentes implantation sont évidemment possibles

- par tableau contiguë
  - non trié (le plus simple)
  - trié
- par liste chaînée
- par hachage

4.9 Adressage associatif

En fait une table n'est rien d'autre que d l'adressage associatif, on recherche l'information v en fonction de l'entrée i, la solution est évidente si i est un nom-

bre, plus compliquée si i est une information quelconque.
Ce qui est rigolo, c'est que l'adressage associatif (pour parler plus clairement, l'adressage par le contenu est quelque fois réalisable électroniquement. Si l'information i n'est pas trop grande, parce qu'elle joue évidemment le rôle d'une adresse, et que les mémoires associatives électroniques existantes ne sont pas encore trés grandes.
Baratin que tout cela ...

4.10 Hachage

Le principe du *hashcoding* est le suivant, on associe par calcul plus ou moins savant, un nombre compris dans un certain intervalle à une information qui n'est à priori pas numérique.

4.11 On mélange tout

L'implantaion d'une table peut être faite en mélangeant toutes ces recettes, on peut imaginer une implantation par hachage où les conflits sont gèrés par une implantation par liste contiguë triée ...
D'où les tables et sous-tables, adressage principal, secondaire etc.. Tout ceci dépend évidemment beaucoup de l'application, de la taille des informations à traitées et de la rapidité d'accés souhaitée.

4.12 Conclusion

Il faut travailler au cas par cas, l'implantation est plus importante que la spécification. Celle est au demeurant trés simple et ne permet pas du tout de guider les choix de l'implantation.

Chapitre 5

Graphes

5.1 Qu'est ce qu'un graphe?

5.1.1 Définition mathématique

Un graphe est le couple $(\mathcal{S}, \mathcal{A})$

où $\mathcal{S}$ est un **ensemble** fini de sommets S et $\mathcal{A}$ une **liste** de triplets (S_o, S_d, V_a)

$$S_o \in \mathcal{S} \qquad S_d \in \mathcal{S} \qquad V_a \in \mathcal{V}_{\mathcal{A}}$$

V_a est la valeur de l'arc qui va du sommet origine S_o au sommet destination S_d.

Notons $\mathcal{V}_{\mathcal{A}}$ l'ensemble des valeurs que peut prendre un arc A.

Attention: la définition telle qu'elle est donnée ici implique que l'on peut avoir deux arcs de même valeur entre deux sommets. C'est pour cela que l'on parle de liste d'arcs et non pas d'ensembles.

5.2 Les différents types de graphes

La définition qui nous venons de donner peut être simplifiée:
- Tous les arcs entre deux sommets sont différents.
- Les arcs peuvent ne pas prendre de valeur, l'arc **est** ou **n'est pas**.

- Il n'y a qu'**un arc entre deux sommets**.
- Les arcs ne sont **pas orientés**.

Les opérations que nous définirons en dépendent évidemment.

5.3 Quelles opérations ?

5.3.1 . . . les élémentaires

adjonction d'un sommet
suppression d'un sommet
adjonction d'un arc entre 2 sommets (avec l'éventuelle valeur)
suppression d'un arc entre 2 sommets
n'importe quel sommet du graphe
n'importe quel arc

5.3.2 . . . les nécessaires

ensemble de sommets
ensemble des arcs issus de S
ensemble des sommets accessibles à partir de S
n'importe quel arc issu de S
ensemble des arcs incidents à S
n'importe quel arc incident à S
etc.

5.3.3 . . . les pratiques.

pour tout sommet de G
pour tout arc issu de S
pour tout sommet atteignable à partir de S
etc.

5.4 Spécification

Il faut les spécifications relatives à l'ensemble des sommets, celles concernant la liste d'arcs avec, entre autres, la précondition

pre(adjonction-arc($(S_1, S_2, V_a)) = S_1, S_2 \in$ sommets(G)

5.5 Implantation

5.5.1 ... par matrice booléenne

5.5.2 ... par ensemble de sommets et listes de triplets

5.5.3 ... par ensemble de sommets et pour chaque sommet une liste de couples (sommet adjacent, valeur de l'arc)

5.5.4 ... mongraphe

Cette implantation est particuliaire en ce sens que j'essaye de définir un graphe par un sommet et une liste de graphes adjacents à ce sommet. Pour simplifier nous nous limiterons au graphe:

- le graphe vide n'existe pas
- le graphe est connexe
- un seul arc non valué entre 2 sommets

Pour ne pas se laisser influencer par des idées toutes faites définissons notre graphe par des mots non significatifs.

LES Objets

Notre graphe est une *Schtroumpferie*, un sommet est un *Schtroumpf*.

Les opérations de base

Soit B,D des *Schtroumpferies*, s un *Schtroumpf*.

On définit

> les valeurs logiques **Tintin** (faux) et **Milou** (vrai)
> **bleu**(s) la *Schtroumpferie* reduite au *Schtroumpf* s.
> **rouge**(B) le sommet de B
> **lucky**(B) est Milou si B est reduit à un seul *Schtroumpf*
> **fagot**(B,D) la *Schtroumpferie* obtenue en reliant la *Schtroumpferie* D au sommet rouge(B)
> **smala**(B) une *Schtroumpferie* liée à rouge(B)
> **qurach**(B,D) on retire le lien vers la *Schtroumpferie* reliée à rouge(B)

programmons la recherche d'un sommet s

On veut

> **ferie**(s, B) qui nous fournit la *Schtroumpferie* du *Schtroumpf* s contenue dans la *Schtroumpferie* B

ferie(s, B)= **si** s ==rouge(B)
alors B
sinon D ==smala(B)
si s ==rouge(D))
alors D
sinon ferie(s,qurach(B, D))

Cette spécification est à creuser, elle me parait intéressante, un graphe est un sommet connecté à d'autres graphes. (. . . à suivre)

5.6 Applications

5.6.1 Le réseau routier

Une agence de voyage dispose d'un graphe **réseau** contenant:

un ensemble de villes ayant chacune un nom
des routes orientées entre les villes avec leur longueur

Une personne de l'agence doit pouvoir:

adjoindre une nouvelle ville, une nouvelle route
avoir accés à l'ensemble des villes, aux routes issues d'une ville donnée
supprimmer une ville ou une route.

Un client de l'agence doit pouvoir:

avoir la liste triée des villes voisines d'une ville donnée
demander le chemin pour aller d'une ville à une autre
éventuellement le chemin le plus court.

On définira précisément chaque opération en veillant à ce que les différents niveaux soient clairement séparés.

5.6.2 Une molécule

Une molécule est faite d'atomes, entre les atomes existent des liaisons.
A chaque atome on attribue un nom (C001, H007 etc.), un type (Carbone, Hydrogène etc.), une position dans l'espace.
Une liaison relie deux atomes, on lui associe sa nature (simple, double ou triple, liaison hydrogène etc.) et sa longueur.

On veut chercher des cycles, déterminer le squelette pour les molécule biologiques. On veut dessiner la molécule en la représentant par des boules colorées et des liaisons en formes de batonnets.

5.7 Circuit logique, réseau cadencé par les données

On se propose de simuler un circuit logique comportant des portes OU, ET, NON ainsi que des fonctions AFFICHE qui permettent de visualiser les valeurs au fur et à mesure qu'elles apparaissent.
Les portes OU et ET ont chacune deux entrées,les portes NON et AFFICHE ont une entrée. Toutes, sauf AFFICHE ont un nombre indeterminé de sorties.
On lance l'exécution du réseau par $FIRE(v, i, p)$ qui consiste à envoyer la valeur logique v sur l'entrée i de la porte p ... et on attend le ou les éventuels résultats de AFFICHE.
Chaque porte devra savoir dans quel état elle se trouve, quelles entrées sont pourvues. Elle *s'exécute* dès que toutes les entrées sont fournies. L'exécution *consomme* les entrées, la porte est prète pour une nouvelle fournée de données.

5.7.1 Quels problèmes peuvent apparaîtrent?

L'existance de circuits n'est pas trés recommandée, en effet, les valeurs émises successives sur la même entrée peuvent ne pas être consommées par la porte réceptrice.

5.7.2 Comment construire le réseau?

- Il faut nommer les fonctions.
- Créer et défaire des liens.
- Réinitialiser tout le réseau (pas simple)

5.7.3 Généralisation

La méthode peut évidemment se généraliser a tout ce qu'on veut:

- Introduire des bascules?
- Les portes sont des fonctions quelconques.
- Les entrées sont gérées en files d'attentes.
- On véhicule autre chose que des booléens.