Par :: Yann Guidon :: whygee@f-cpu.org ::

GDUPS ou la chasse aux fichiers dupliqués

Voici la description d'un petit programme bien utile, qui permet de s'y retrouver dans une arborescence contenant des fichiers en multiples exemplaires ou aux noms identiques. Ce n'est pas un outil de nettoyage mais un assistant permettant de réduire l'entropie d'un disque dur. Pour les retardataires des articles d'Yves Mettier ou pour les amateurs, nous allons passer rapidement en revue plusieurs techniques de programmation qui font partie de la culture de base. Mises bout à bout, ces briques algorithmiques donnent un programme qui sent bon l'UNIX et qui devrait intéresser les programmeurs, même débutants.

Introduction

D'année en année, la profusion des logiciels et le butinage intensif sur Internet créent un nombre croissant de fichiers aux noms ou aux contenus identiques, et un même fichier peut parfois se retrouver dupliqué à plusieurs endroits sans que l'on ne s'en aperçoive. Les utilitaires `find` et `locate` permettent de localiser un fichier dont nous connaissons le nom ou une partie, mais que faire lors d'une recherche plus générale ?

Comment comparer des répertoires pour éviter des collisions de noms ou de contenu, lors de sauvegardes, de copies ou d'effacements ? Est-il possible de *merger* rapidement des développements séparés d'un même programme source (sans `cvs`) ? En résumé, comment réduire l'entropie dans les fichiers, au cas par cas ?

La solution "unixienne" et rapide à ce problème passe souvent par l'écriture d'un script `shell`.

Nous pourrions, simplement, comparer la sortie de la commande `ls -al` avec toutes les sorties précédentes. Toutefois, il faudrait être patient car l'ordinateur va *forker* $N^2/2$ fois pour comparer N fichiers, ce qui prendrait des proportions inadmissibles si un disque entier doit être analysé. Notre programme, appelé "`GDUPS`" pour "*GNU Duplicates*", utilise un algorithme plus rapide qui effectue seulement $N*\log_2(N)$ comparaisons.

Il va aussi plus loin que la recherche des doublons exacts : il indique les fichiers dont le nom ou le contenu sont identiques, nous laissant libres d'opérer le type de nettoyage désiré.

La première partie du programme balaie récursivement le répertoire spécifié et dresse deux listes, une pour tous les noms des fichiers rencontrés, et une deuxième (en parallèle) pour les tailles de ces fichiers.

Si des tailles sont identiques, le programme calcule la signature de tous les fichiers concernés, ce qui permet de s'y retrouver facilement lorsque plus de deux fichiers ont une taille identique.

Le programme utilise en moyenne une centaine d'octets de mémoire par fichier. Il se termine en imprimant un rapport en clair pour l'utilisateur, facilement filtrable par `grep` par exemple.

Principes de base

Pour réduire la charge de travail de l'ordinateur, la première étape consiste à utiliser des appels systèmes au lieu de lancer un programme pour chaque opération. Le programme sera monolithique et moins flexible mais il s'exécutera plus rapidement. La deuxième étape est la mise au point d'une sorte de "base de données" qui sera construite et consultée à mesure que les fichiers seront balayés. Toujours pour simplifier le programme, nous n'utiliserons pas de programme externe et nous examinerons quelques techniques au passage. Puisque la "base de données" contiendra au maximum N fichiers, nous pouvons déjà évaluer que sa consommation de mémoire sera proportionnelle à N. Sur la base de 100 octets par fichier en moyenne et une RAM de 128Mo, il faudra analyser un système de fichiers contenant plus d'un million de fichiers avant que le programme ne commence à *swapper*.

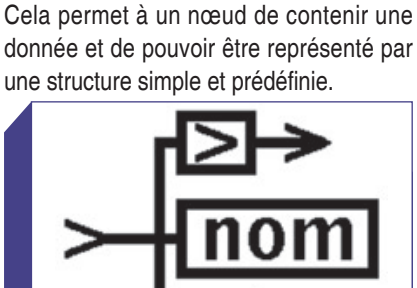
L'organisation des données en arbres binaires (il y en aura deux, un pour les noms et un autre pour les tailles) permet de comparer une valeur à N autres valeurs en $\log_2(N)$ étapes. Puisqu'il y aura au maximum N comparaisons (une pour chaque nouveau fichier ajouté à nos listes), notre programme effectuera au maximum $N*\log_2(N)$ étapes, au lieu des N^2 étapes prévues initialement.

Une autre organisation aurait pu être utilisée à la place d'arbres. Les *hashtables* (tables de hachage) effectuent une correspondance directe entre une *clé* (le nom ou la taille du fichier) et une adresse en mémoire par l'intermédiaire d'un tableau.

Cependant, comme nous ne connaissons pas d'avance le nombre d'éléments à tester, il est impossible de déterminer la taille du tableau de hachage. Il faut noter qu'il est compliqué de changer la taille d'une table de hachage au cours de son utilisation, contrairement à un arbre. Enfin, l'optimalité de cette solution n'est pas critique puisque le programme passe en pratique beaucoup plus de temps à calculer les signatures.

Description d'un arbre de recherche

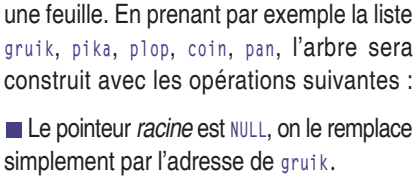
L'arbre que nous utiliserons s'apparente à l'arborescence du système de fichiers que nous connaissons bien : l'arbre contient la racine (*root*), les sous-arbres (répertoires) et les feuilles (fichiers). Cependant nous allons restreindre fortement la structure : un "répertoire" ou *noeud* ne peut contenir qu'un seul fichier (obligatoire) et deux "sous-répertoires" ou liens vers d'autres nœuds. Cela permet à un nœud de contenir une donnée et de pouvoir être représenté par une structure simple et prédéfinie.



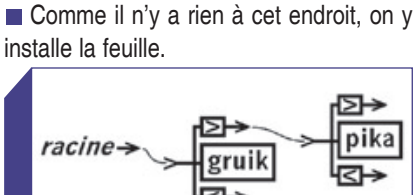
Nous obtenons une structure qui peut être balayée avec un programme très simple. Par analogie à un parcours en voiture, nous voyageons de nœud en nœud en nous posant seulement deux questions pour chaque nœud (ou embranchement de la route) : sommes-nous arrivés à destination, et si non, faut-il aller à gauche ? Les réponses déterminent si l'on reste sur place, s'il faut aller à gauche ou à droite.

Pour construire un arbre, on répète l'opération de recherche pour chaque nouvel élément, que l'on insère quand on arrive à une feuille. En prenant par exemple la liste `gruik`, `pika`, `plop`, `coin`, `pan`, l'arbre sera construit avec les opérations suivantes :

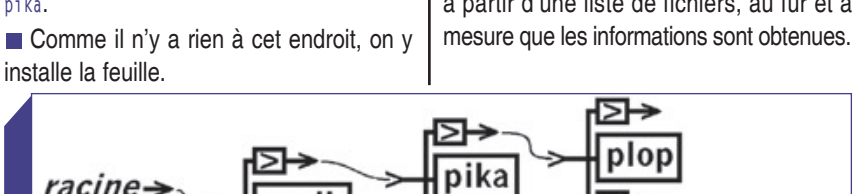
- Le pointeur *racine* est `NULL`, on le remplace simplement par l'adresse de `gruik`.



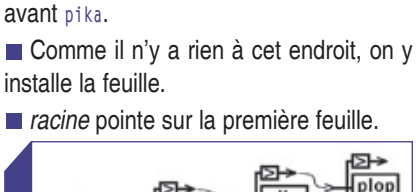
- *racine* pointe sur la première feuille.
- `pika[0] > gruik[0]` donc on place `pika` à une position supérieure.
- Comme il n'y a rien à cet endroit, on y installe la feuille.



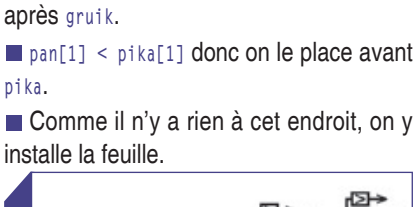
- *racine* pointe sur la première feuille.
- `plop[0] > gruik[0]` donc on se déplace après `gruik`.
- `plop[1] > pika[1]` donc on le place après `pika`.
- Comme il n'y a rien à cet endroit, on y installe la feuille.



- *racine* pointe sur la première feuille.
- `coin[0] < gruik[0]` donc on le place avant `pika`.
- Comme il n'y a rien à cet endroit, on y installe la feuille.
- *racine* pointe sur la première feuille.



- `pan[0] > gruik[0]` donc on se déplace après `gruik`.
- `pan[1] < pika[1]` donc on le place avant `pika`.
- Comme il n'y a rien à cet endroit, on y installe la feuille.



Si nous trouvons un élément semblable, nous pouvons rajouter l'élément courant à une liste chaînée qui peut contenir d'autres éléments similaires. Toutefois, afin d'accélérer la recherche d'éléments semblables entre les différents arbres, nous pouvons encore utiliser un autre arbre qui va trier les fichiers selon un autre critère.

Construction pratique d'un arbre

Dans notre programme, nous n'allons pas "placer" les feuilles, ce n'est qu'une vision abstraite. En réalité, les pointeurs permettent de s'affranchir complètement d'une quelconque topologie. L'arbre sera construit à partir d'une liste de fichiers, au fur et à mesure que les informations sont obtenues.

La contrainte principale concerne l'occupation mémoire : notre "nœud" décrivant un fichier contiendra seulement les informations nécessaires, on ne va pas garder ou recopier tout ce que le noyau nous renvoie.

De plus, deux arbres sont créés avec les mêmes données : les "nœuds" sont utilisés pour les deux arbres, mais il faut éviter toute duplication. Ils vont donc contenir les pointeurs nécessaires aux deux arbres, et on n'aura besoin de créer chaque nœud qu'une seule fois. Ainsi, on peut créer deux arbres entrelacés ou plus à partir d'une seule liste, il suffit pour chaque nœud d'allouer une paire de pointeurs.

Balayage des fichiers

Il existe plusieurs moyens d'obtenir une liste de fichiers. La première consiste à exécuter la commande "`find .`" et en analyser la sortie, après l'avoir redirigée vers un descripteur de fichier préalablement préparé. Cela permet de filtrer les fichiers par la commande `grep` par exemple.

Toutefois, même si cette première méthode est tout à fait unixienne par la flexibilité qu'elle permet, l'utilisation d'un programme externe peut poser des problèmes de portabilité, de mémoire et de vitesse. Une autre solution serait d'utiliser la base de données de `locate` (voir "`man 5 locatedb`") mais il en existe au moins deux versions que nous ne pouvons pas examiner ici. Nous pouvons remarquer que la commande `updatedb` (qui met à jour la table de `locate`) utilise `find`.

Finalement, la solution la plus directe est "manuelle" : le source `liste.c` montre comment utiliser `opendir()`, `readdir()` et `stat()` pour lister le contenu d'un répertoire.

```
   (extrait de liste.c)
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <dirent.h>

DIR *dir;
struct direct *file;
struct stat state;

int main() {
  /* 1 : ouvrir le répertoire */
  if ((dir=opendir("."))==NULL) {
    fprintf(stderr,"can't open directory\n");
    return 2;
  }

  /* 2 : y lire les entrées tant qu'il y a des entrées */
  while ((file=readdir(dir))!=NULL) {
    /* déterminer le type de fichier */
    if (stat(file->d_name,&state)==0) {
      if (S_ISREG(state.st_mode))     /* fichier normal */
        printf("f: %s\n",file->d_name);
      else {
        if (S_ISDIR(state.st_mode))   /* répertoire */
          printf("D: %s\n",file->d_name);
      }
    }
  }
  return 0;
}
```

Après quelques modifications, on arrive à `explore.c` qui permet de lister récursivement le contenu des sous-répertoires et d'en extraire les caractéristiques, comme la taille du fichier, son type ou ses droits d'accès (voir `man [l]stat()`). La complication majeure vient de la gestion de la récursivité et du chemin.

On crée d'abord une fonction récursive, appelée par le `main()` puis par elle-même lorsqu'un répertoire est trouvé par un appel à `lstat()`. Pour éviter de partir dans une boucle infinie éventuelle, `lstat()` est utilisé à la place de `stat()` qui suit les liens symboliques. La pile, elle, permet de gérer implicitement et sans effort un nombre (inconnu au départ) de variables simultanées.

Le code va procéder autrement pour mémoriser le chemin, puisqu'il peut devenir très long et il a une taille variable. Il n'est pas mémorisé dans la pile, mais dans une variable globale qui est modifiée par la fonction récursive. Notre code n'utilise qu'un seul process et un seul processeur, et un seul niveau d'appel n'est possible à la fois, cela ne pose donc aucun risque de collision.

Chaque niveau d'appel de la fonction va donc mémoriser la taille précédente de la chaîne, ajouter le nouveau chemin, et l'invalider (le restaurer) à la fin. Par contre, la taille limite du chemin est arbitraire et risque de poser quelques soucis à un moment ou un autre, veuillez donc considérer ce code comme une illustration d'un principe avant de menacer d'envoyer des lettres de délation à RMS.

```
   (extrait de explore.c)
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <stdlib.h>

#define MAXPATHSIZE    255

/* Les données suivantes sont globales pour réduire l'occupation
   de la pile et éviter des opérations redondantes */
int taille_chemin;
struct stat state;
char chemin[MAXPATHSIZE];

void explore_recursif() {
  /* variables locales : elles sont allouées à chaque appel
     et elles peuvent donc exister en de multiples exemplaires */
  DIR *dir;
  int temp_taille_chemin, taille_nom;
  struct direct *file;

  /* 1 : ouvrir le répertoire */
  if ((dir=opendir(chemin))==NULL) {
    fprintf(stderr,"impossible d'ouvrir %s\n",chemin);
    return;
  }

  temp_taille_chemin=taille_chemin; /* sauvegarde */
  chemin[taille_chemin++]='/';
  chemin[taille_chemin]=0;
  printf("ouverture de %s\n",chemin);

  /* 2 : y lire les entrées */
  while ((file=readdir(dir))!=NULL) {
    /* tant qu'il y a des entrées */

    /* filtrage économique de "." et ".." */
    if ((file->d_name[0] == '.')
      && ((file->d_name[1] == 0)    /* "." */
        || ((file->d_name[1] == '.')
          &&(file->d_name[2] == 0)))) { /* ".." */
      /* ne rien faire */
    } else {

      /* concaténation (déplacée avant le stat() pour pouvoir
         interroger n'importe quel fichier hors du répertoire courant */
      taille_nom=strlen(file->d_name);
      taille_chemin=temp_taille_chemin+taille_nom+1;
      if (taille_chemin > MAXPATHSIZE) {
        fprintf(stderr,"\n erreur : chemin trop long\n");
        exit(3);
      }
      memcpy(&chemin[temp_taille_chemin+1],file->d_name,taille_nom+1);

      if (lstat(chemin,&state)==0) {    /* déterminer le type de fichier
*/
        if (S_ISREG(state.st_mode))    /* fichier normal */
          printf("   %s (%ld)\n",chemin,state.st_size);
        else {
          if (S_ISDIR(state.st_mode))  /* répertoire */
            explore_recursif();
        }
      }
    }
  }
  closedir(dir);
  taille_chemin=temp_taille_chemin; /* restaure */
  chemin[taille_chemin]=0;          /* nettoie */
  printf("fermeture de %s\n",chemin);
}

int main() {
  /* lit le chemin courant */
  if (getcwd(chemin,MAXPATHSIZE)==NULL)
    return 1;
  /* initialise la variable glogale */
  taille_chemin=strlen(chemin);
  explore_recursif();
  return 0;
}
```

A partir de ce code, nous pouvons commencer à construire notre "base de donnée" dans le format qui nous convient le mieux.

Allocation de la mémoire

Puisque nous allons manipuler un nombre incertain et potentiellement grand d'éléments de taille variable, nous ne pouvons pas facilement utiliser de tableau statique, dont la taille est prédéterminée. Notre programme doit gérer une structure dynamique et allouer de la mémoire à la demande : l'utilisation de `malloc` est nécessaire. Pour cette occasion, rappelons que `malloc()` a la réputation d'être mal conçu : dans notre cas, il va probablement gâcher les ressources de la machine. Par exemple, l'appel système est plus lourd qu'un simple appel à une fonction, la récupération de la mémoire est inexistante (le *garbage collecting* a été oublié) et la granularité est

inadaptée (par exemple, il va allouer une page de 4Ko même si on lui demande juste 100 octets). Bien sûr cela varie fortement avec les versions de systèmes d'exploitation mais il est préférable de s'en méfier. La solution est de programmer un petit "wrapper", une procédure qui va découper en petits morceaux un gros bloc alloué avec `malloc()`.

Le principe est simple et heureusement pour la concision, nous n'avons pas besoin de libérer les morceaux alloués, donc de garder en mémoire l'adresse de tous les blocs alloués : un seul pointeur et un "indicateur de niveau" suffisent.

Lors de l'allocation "fine", la seule contrainte est d'arrondir les blocs sur 8 octets pour que certaines parties des structures soient correctement alignées (la taille des fichiers par exemple).

La taille des "gros" blocs alloués à chaque fois par `malloc()` est définie par `TAILLE_BLOC` et peut être changée si nécessaire, elle correspond ici aux besoins moyens d'un disque dur.

Le wrapper ressemble à ceci :

```
#include <stdlib.h>     /* pour malloc */

#define TAILLE_BLOC 524288
/* 512KO devraient suffire pour avoir peu de blocs si on scanne
   tout un disque dur. Réduire ou augmenter si nécessaire. */

/* Variables globales pour le wrapper de malloc : */

/* nombre de blocs alloués et d'appels à mon_malloc
   (juste pour les statistiques) : */
int nb_blocs=0, nb_elements=0;

/* pointe sur le dernier bloc alloué : */
char * bloc_courant;

/* nombre d'octets occupés dans le bloc : */
int index_bloc_courant=TAILLE_BLOC; /* astuce : il est
  initialisé afin de déclencher une nouvelle
  allocation dès le premier appel à mon_malloc() */

/* Gestion fine de l'allocation de la mémoire (sans libération) : */
void * mon_malloc(int taille) {
  void *temporaire;
  nb_elements++;
  taille=(taille+7) & -8; /* arrondit la taille aux
     8 octets supérieurs afin d'éviter toute faute d'alignement */

  index_bloc_courant+=taille;
  if (index_bloc_courant > TAILLE_BLOC) {
    /* le bloc courant est saturé, on en alloue un autre */
    bloc_courant=malloc(TAILLE_BLOC);
    if (bloc_courant == NULL) {
      fprintf(stderr,"\n erreur de malloc\n");
      exit(5);
    }
    nb_blocs++;
    index_bloc_courant=0;
  }

  temporaire=bloc_courant+index_bloc_courant;
  index_bloc_courant+=taille;
  return temporaire;
}
```

On remarquera l'astuce consistant à initialiser `index_bloc_courant` à une valeur supérieure à la taille d'un bloc.

L'ensemble est donc *autoinitialisant*, il n'y a pas besoin d'appeler ou de programmer une fonction d'initialisation avant d'utiliser `mon_malloc()` la première fois.

Structure des descripteurs

La structure la plus importante est celle qui décrit un fichier (`struct noeud_fic`). Il faut y consigner ses caractéristiques (dont la taille) et tous les pointeurs vers les branches des arbres qui vont classer le descripteur par la taille et l'ordre alphabétique.

Nous avons remarqué que nous n'avons pas besoin de libérer la mémoire durant le programme, nous pouvons donc nous passer de la mémorisation des pointeurs d'allocation.

Même avec cette économie, un descripteur de fichier utilise déjà 40 octets.

```
/* description d'un fichier : */
struct noeud_fic {
  /* propriétés/caractéristiques du fichier : */
  off_t taille;  /* taille du fichier en octets  */
  time_t date; /* défini par <time.h> et <bits/types.h> */
  long int CRC; /* signature, ignorée si arbre_taille_pareil==NULL */
  struct noeud_chemin *chemin; /* pointeur vers le nom du chemin */

  /* arbres entrelacés : */
  struct noeud_fic
    *arbre_taille_prec,
    *arbre_taille_pareil,
    *arbre_taille_suiv,
    *arbre_nom_prec,
    *arbre_nom_pareil,
    *arbre_nom_suiv;

  /* partie à taille variable du noeud, située
     obligatoirement à la fin de la structure : */
  unsigned char nom[0];      /* nom du fichier */
};
```

La récursivité

- La récursivité est souvent utilisée conjointement à des arbres ou d'autres structures "autosimilaires". C'est une solution pratique pour calculer des fractales comme le "*flocon de von Koch*" ou explorer des arborescences.
- Il faut rappeler que l'utilisation de la pile est moins souvent efficace qu'une gestion manuelle et soignée de la mémoire : chaque appel de la fonction à elle-même nécessite de sauvegarder de nombreuses données redondantes. Avant de créer un programme utilisant la récursivité, il faut donc se demander s'il est possible de transformer le programme en boucle "`do ... while`".
- Il existe en fait deux types de récursivité : la récursivité de programme (pour certains programmes complexes) et la récursivité de données (très répandue). Par exemple, le balayage complet d'un arbre est de nature récursive alors que la recherche dans celui-ci est purement itératif.
- Le choix entre données globales et locales est aussi déterminant si la complexité devient importante. Le programme donné en exemple reste simple afin de faire découvrir les différentes options.

Les arbres

■ Les arbres ou arborescences sont des structures très répandues et pratiques en informatique. D'autres structures utiles sont les listes chaînées et les hashtables qui sont d'ailleurs souvent utilisées en même temps que les arborescences.

■ On utilise les arbres par exemple pour les systèmes de fichiers, pour analyser les programmes ou représenter des expressions mathématiques. C'est un outil algorithmique qui permet de réduire la complexité ou le temps d'exécution d'un programme lorsqu'il est utilisé correctement. Certains ordinateurs massivement parallèles comme la Connexion Machine 5 (fabriquée par la Thinking Machines Corporation, aujourd'hui disparue) sont d'ailleurs construits autour de ces algorithmes. Les jeux vidéo utilisent aussi la technique présentée ici pour trier les facettes affichées en 3D.

■ Si une feuille de l'arbre est confondue avec une autre, on ne peut plus parler d'arbre, mais de graphe et l'algorithme de balayage ou d'exploration est différent. Par exemple, lorsqu'on crée un alias sur un fichier ou un répertoire, l'exploration récursive de l'arbre fait apparaître plusieurs fois les mêmes éléments alors qu'il n'existe qu'un exemplaire de chaque. Si le graphe devient trop complexe, l'exploration récursive ne peut pas fonctionner car elle peut tomber sur une boucle. C'est une des raisons pour lesquelles il faut rester prudent lorsque l'on utilise les alias. De plus, `lstat()` est utilisé ici à la place de `stat()` pour éviter une boucle infinie lors de l'exploration récursive.

Le descripteur contient une partie variable et extérieure pour stocker le nom du fichier, ce qui permet d'utiliser une méchante astuce de langage C. Il est en effet possible de définir une chaîne de caractères de taille nulle, et le compilateur (selon sa configuration) ne vérifie pas si les bornes sont dépassées. Nous pouvons donc, en faisant attention, définir une chaîne de caractères de taille variable, mais qui reste utilisable tout à fait normalement, sans utiliser la syntaxe obscure des pointeurs.

Il faut toutefois empêcher le compilateur de vérifier l'intervalle des indices. De plus, cette astuce est possible ici puisqu'il n'y a qu'une seule zone de taille variable, qui est située à la fin de la structure et n'empiétant pas sur d'autres données. Enfin, on tient toujours compte de la taille de la chaîne lors de l'allocation de la mémoire.

Tri par insertion sur la longueur des noms

Une autre astuce consiste à mélanger la structure des chaînes de type Pascal et C. Pour rappel, une chaîne de caractères en C est de type ASCIIZ : ce sont des codes ASCII dont la fin est délimitée par un octet à Zéro. Une chaîne Pascal, elle, commence par un octet qui indique combien de caractères sont dans la chaîne. Cela limite le nombre de caractères d'un nom de fichier à 255 mais cette limite est très rarement dépassée. Le mélange de ces conventions crée une chaîne commençant par un caractère de taille et terminée par un zéro.

Par exemple, la chaîne "`.bashrc`" sera codée avec la valeur 7 à la position 0, suivie de 7 caractères alphabétiques et terminée par un zéro. Le tout sera codé très simplement avec `taille = nom[0]` et pour imprimer le nom d'un fichier pointé par `t : printf(&(t->nom[1]));` (ce qui est légèrement plus lisible que `printf(&(t->nom)+1);`).

L'avantage ? Cette représentation accélère beaucoup la comparaison de chaînes avec `str(n)cmp` ! La fonction va s'arrêter tout de suite si la taille de la chaîne ne correspond pas. Ce dégrossissage est bien pratique pour que l'arbre ne se construise pas de manière trop déséquilibrée, ce qui réduirait la vitesse d'exploration de l'arbre.

Prenons un exemple : des fichiers ont été préalablement déplacés par la commande "`mv * destination`" et nous nous trouvons justement dans le répertoire `destination`. Le *joker* `*` va être remplacé par le *shell* par la liste triée de tous les fichiers du répertoire courant, et leur entrée respective va être inscrite dans l'ordre alphabétique sur le système de fichiers.

Maintenant, explorons le répertoire avec `readdir()` : il va retourner, au fur et à mesure des appels, les noms des fichiers dans l'ordre alphabétique.

Si nous construisions l'arbre directement à partir de ces données, l'arbre deviendrait complètement déséquilibré : les noms pré-triés obligeraient l'algorithme d'exploration de l'arbre à toujours emprunter le même chemin qui s'allongerait de plus en plus dans la même direction.

L'arbre revenant à une structure linéaire, son efficacité serait similaire à une liste chaînée (sans ses avantages) et le temps d'élaboration total serait de l'ordre de `N2/2`...

Tri alphabétique

Liste de départ : `coin`, `gruik`, `pan`, `pika`, `plop`

```
1) (racine)==>[coin]

2) (racine)==(?)==>[coin]
            \==(+)==>[gruik]

3) (racine)==(?)==>[coin]
            \==(+)==(?)==>[gruik]
                        \==(+)==>[pan]

4) (racine)==(?)==>[coin]
            \==(+)==(?)==>[gruik]
                        \==(+)==(?)==>[pan]
                                    \==(+)==>[pika]

5) (racine)==(?)==>[coin]
            \==(+)==(?)==>[gruik]
                        \==(+)==(?)==>[pan]
                            \==(+)==(?)==>[pika]
                                        \==(+)==>[plop]
```

Effectuer le tri en commençant par la taille du nom va redistribuer les informations, rendant l'arbre explorable un peu plus rapidement sans recourir à des algorithmes complexes de rééquilibrage dynamique.

C'est donc un "*hack*" simple, une heuristique non optimale mais qui économise des efforts de codage et de temps d'exécution.

Tri alphabétique modifié

Liste de départ : `4coin`, `5gruik`, `3pan`, `4pika`, `4plop`

```
1) (racine)==>[4coin]

2) (racine)==(?)==>[4coin]
            \==(+)==>[5gruik]

3)          /==(-)==>[3pan]
   (racine)==(?)==>[4coin]
            \==(+)==>[5gruik]

4)          /==(-)==>[3pan]
   (racine)==(?)==>[4coin]
            \          /==(-)==>[4pika]
             \==(+)==(?)==>[5gruik]

5)          /==(-)==>[3pan]
   (racine)==(?)==>[4coin]
            \            /==(-)==(?)==>[4pika]
             \          /          \==(+)==>[4plop]
              \==(+)==(?)==>[5gruik]
```

Cela permet aussi (c'était l'idée initiale) de regrouper dans le rapport des fichiers d'une même série (différentes versions d'un fichier ou différentes révisions d'un logiciel par exemple) car ils ont souvent un nom de longueur identique dont ne changent que les dernières lettres.

Comparaison des fichiers

Lorsque nous détectons que deux fichiers ont la même taille, cela ne garantit pas qu'ils soient identiques. Seule une comparaison directe (bit à bit) peut confirmer la correspondance.

C'est facile à faire pour deux fichiers, mais pour plusieurs fichiers, il faut tous les comparer entre eux, ce qui nécessite encore `(N²-N)/2` opérations.

Nous pouvons réduire le champ de possibilités en calculant une signature pour chaque fichier suspect. Même si la signature a une chance sur quatre milliards d'être fausse (une "collision" n'est jamais impossible), nous pouvons au moins être sûr qu'il existe une différence si les signatures ne correspondent pas.

Nous réduisons déjà considérablement les chances d'erreur si la signature et la taille du fichier ne sont pas identiques.

Cependant, la signature d'un fichier n'est pas une fonction fournie par le noyau ou le système de fichiers.

Nous devons choisir une méthode relativement fiable, rapide et portable. Pas besoin d'une signature `MD5`, trop lourde puisque nous n'avons pas besoin d'une fiabilité cryptographique.

Plus simplement, on peut utiliser un algorithme de `CRC` ("*Cyclic Redundancy Check*"), couramment utilisé pour détecter les erreurs dans les fichiers ou les flux de données.

Une petite recherche sur Internet donne de nombreuses références dans le domaine public, dont l'excellent texte de *Ross N. Williams*, "*A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS*".

C'est une référence incontournable et facilement compréhensible (pour les anglophones) disponible à **http://www.ross.net/crc/crcpaper.html**. Le fichier `signature.c` implémente l'algorithme `CRC32`, aussi employé pour les paquets `Ethernet` ou par `PKZIP` par exemple. Les premières versions de `gdups` utilisaient une table de `CRC` précalculée. Celle-ci est difficile à obtenir et occupe beaucoup de place, ne serait-ce que dans ces pages. J'ai donc créé une nouvelle version des programmes générant la table de `CRC` au démarrage.

La fonction des tables de `CRC` ainsi que leur fabrication sont expliquées par le texte de *Ross Williams* mais je n'ai pas recopié son code directement : le cas de `CRC32` utilise une version *inversée* du code de précalcul qu'il utilise.

Pour réduire la taille du code dans notre cas particulier, j'ai donc inversé le polynôme (une constante) pour éviter d'inverser les données de travail, de même que la condition de test : on ne teste plus le `MSB` mais le `LSB` (le bit 0). Tout le reste, c'est à dire le code de signature, est standard.

```
/*
  fichier : signature2.c simplifié
  (c) Yann Guidon 2000 (whygee@f-cpu.org)
  créé le mardi 17 octobre 2000
  version jeudi 5 mars 2004

Ce code calcule la signature d'un fichier
avec un code CRC32 (un peu modifié).

Pour compiler :
   gcc signature2.c
*/

#include <sys/stat.h>
#include <unistd.h> /* pour lstat() */
#include <stdio.h>  /* pour fprintf() */
#include <stdlib.h> /* pour malloc() */

#define CRC_POLY_REV 0xEDB88320 /* poly CRC32 inversé
   CRC32 classique : polynôme = 0x04C11DB7
      (32,26,23,22,16,12,11,10,8,7,5,4,2,1,0) */

unsigned long int CRC32tab[256],
 CRC_val, CRCSeed = 0xffffffff;

#define CRC_BUFFER_CHUNK_SIZE 65536 /* peut être changé */
unsigned char *CRC_buffer=NULL;

/* calcule la table de CRC32 */
void mkCRCtab() {
  int i, j;
  unsigned long r;

  for (i=0; i<256; i++) {
    r = i;
    for (j=0; j<8; j++) {
      if ( r & 1 )
         r = (r >> 1) ^ CRC_POLY_REV;
      else
         r>>=1;
    }
    CRC32tab[i] = r;
  }
}

/* calcule le CRC d'un fichier */
void calcule_CRC(char *nom_fic){
  unsigned char *c;
  unsigned int i,taille;
  FILE *in;

  /* init_CRC(); */
  if (CRC_buffer==NULL) {
    /* initialisation du buffer */
    CRC_buffer=malloc(CRC_BUFFER_CHUNK_SIZE);
    if (CRC_buffer==NULL) {
      fprintf(stderr,"\nerreur de malloc() dans calcule_CRC\n");
      exit(6);
    }
  }
  CRC_val=CRCSeed;

  in=fopen(nom_fic,"rb");
  if (in==NULL) {
    fprintf(stderr,"CRC impossible, lecture de %s refusée          \n",nom_fic);
    return;
  }
  fprintf(stderr,"CRCing %s%c                          ",nom_fic,0xD);

  while ((taille=fread(CRC_buffer,1,CRC_BUFFER_CHUNK_SIZE,in))!=0) {
    c=CRC_buffer;
    while (taille--) {
      i=(CRC_val&255)^(*c++);
      CRC_val=CRC32tab[i]^(CRC_val>>8);
    }
  }
  fclose(in);
}

/* banc de test : */
int main(int argc, char * argv[]) {

  if (argc!=2) {
    fprintf(stderr,"mauvais arguments\n");
    exit(7);
  }
  mkCRCtab();
  /* lance le calcul : */
  calcule_CRC(argv[1]);
  printf("signature : %lX\n",CRC_val);
}
```

Pour ajouter un témoin d'avancement, le code de `gdups` affiche une petite chaîne de caractères cyclique à l'intérieur de la boucle `while(fread())`. On peut voir ainsi la proportion du temps passé à calculer une signature par rapport à l'exploration des sous-répertoires.

Affichage du rapport

Lorsque nous détectons qu'un fichier a le même nom ou la même taille qu'un autre déjà mémorisé, il est possible d'afficher directement un message à l'écran. Cependant, en pratique, les messages seront difficiles à interpréter. Il serait intéressant de les regrouper, ou même de les trier pour faciliter l'analyse rapide des cas de figure. Les procédures d'affichage du rapport d'analyse sont aussi des procédures récursives, ce qui garantit que les éléments sont affichés dans l'ordre. Leur niveau terminal balaie les listes chaînées qui ont été constituées et triées lors de la découverte de doublons.

Au final, `gdups` utilise 4 structures arborescentes, associées aux algorithmes qui les traitent :

- L'arbre du système de fichiers, exploré récursivement dans la première passe;
- L'arbre des chemins, balayé uniquement des feuilles vers la racine;
- L'arbre des fichiers triés par nom, préhaché à la racine et terminé par une liste chaînée pour les éléments identiques;
- L'arbre des fichiers triés par taille, éventuellement terminé à chaque feuille par une liste chaînée triée par valeur de `CRC`.

Améliorations et performances

Nous pouvons remarquer que la performance du programme dépend beaucoup du rapport entre la vitesse du disque et celle du processeur couplé à sa mémoire. Les temps d'accès aux fichiers ralentissent énormément le programme qui occupe moins de 25% du temps CPU. Le processeur peut donc continuer à

"philosopher" pendant que le disque dur crépite. Mais que se passerait-il si toutes les données étaient déjà cachées en mémoire centrale ? L'accélération est d'un ordre de grandeur mais peut-on faire mieux ?

Cette première version de `GDUPS` repose sur le principe que le nom et la taille des fichiers sont répartis aléatoirement, ce qui permet à l'algorithme arborescent de fonctionner, on l'espère, en `N*log(N)`. Toutefois, on peut tomber sur des cas particuliers où les heuristiques utilisées sont inefficaces, malgré le pré-hachage sur la taille du nom par exemple. La distinction entre majuscules et minuscules peut être déterminante, il faudrait penser à une version qui ne tient pas compte de la casse des caractères. L'arbre doit pouvoir aussi être "rééquilibré" lorsque sa géométrie rend son balayage inefficace. Il serait utile de connaître la date de dernière modification des fichiers dont on détecte la similarité. On pourrait peut-être laisser le programme fabriquer lui-même des liens symboliques pour remplacer les fichiers en double mais ce serait potentiellement dangereux.

La partie la plus complexe est le rééquilibrage. C'est une opération lourde car elle traite tous les `N` éléments en mémoire, elle doit donc être déclenchée sporadiquement pour ne pas prendre plus de temps que la recherche elle-même. Nous pouvons l'autoriser lorsque l'arbre contient `2^(N*2)` éléments, avec `N>3`, soit pour 256, 1024, 4096 éléments, etc. Il est possible de réorganiser l'arbre de manière plus sophistiquée, par exemple en examinant des sous-arbres qui contiennent moins d'éléments, mais examinons d'abord le cas le plus simple.

L'algorithme pourrait se dérouler en deux étapes et utiliser une représentation intermédiaire. Il va falloir d'abord "lire" l'arbre récursivement pour constituer une liste de pointeurs correctement triés, puis reconstruire l'arbre en relisant la liste (pas besoin de récursivité cette fois-ci). Nous pouvons connaître à l'avance la taille de la liste et utiliser `malloc()` et `free()` sans états d'âme.

Avant d'en arriver là, il faudrait que les opérations du disque dur soient plus rapides. Le calcul de la signature prend encore tellement de temps que l'optimisation du rééquilibrage n'est pas nécessaire à notre niveau. Pour vous en convaincre, voici le résultat de la commande `time` sur un *vieux* Pentium pour scanner tout un disque dur :

```
real    21m8.185s
user    1m34.330s
sys     3m5.120s
```

Les performances sont bien meilleures sur les ordinateurs plus récents, grâce en particulier à la réduction du temps de déplacement des têtes de lecture. Une amélioration future consisterait à trier l'ordre d'accès aux fichiers pour réduire le temps perdu lors de la recherche des secteurs.

Le disque dur consomme au moins les trois quarts du temps total d'exécution du programme pour générer plus de 5Mo de données, scanner 112203 descripteurs de fichiers stockés dans 12Mo de RAM. Et encore, le temps de calcul des signatures n'est pas discernable dans le temps "`user`".

Pour les très gros fichiers, il est certain que l'accélération serait encore plus significative si on limitait la portée du *checksum*. Normalement, si les 64 premiers Ko de deux fichiers donnent une signature différente, il n'y a pas besoin de continuer la vérification sur les mégaoctets restants. La vérification complète aura lieu si les `CRC` partiels correspondent. Cette amélioration est particulièrement souhaitable si, comme moi, vous traitez de gros fichiers multimédia : `gdups` perd son temps à calculer la signature de toutes les versions de fichiers `.WAV` qui, comme par hasard, ont la même longueur.

Une autre amélioration consisterait à pouvoir comparer deux répertoires pour vérifier que leur contenu est identique, ou bien déterminer à quel point ils sont semblables. Actuellement, le rapport final ne présente pas les informations de manière adéquate pour cela, mais elles sont disponibles sous une forme qu'il faut exploiter. A défaut, `diff -r` remplit la même fonction.

Listes chaînées ou arbres

Dans la plupart des programmes, les listes chaînées peuvent être aisément améliorées et remplacées par des arbres binaires. Le programme devra être légèrement modifié mais il fonctionnera généralement bien plus vite qu'avec des listes chaînées, surtout lorsque le nombre d'éléments à manipuler est très grand.

Un bon exemple de cette modification est un système de fichiers classique : la recherche d'un élément est accélérée par une recherche par dichotomie à la place d'une recherche séquentielle. Un système de fichier de serveur peut contenir des centaines de milliers et parfois même des millions de fichiers, il est évident qu'un algorithme adapté est nécessaire puisque le temps d'accès aux fichiers sur les disques durs ne diminue que très lentement.

Toutefois, quelle que soit la solution employée (structures à accès linéaire ou arborescent), il ne faut pas oublier que les techniques de bases doivent être maîtrisées. L'allocation de la mémoire ou la gestion des structures de base sont autant de détails qui jouent sur l'efficacité et la facilité de programmation.

D'une manière plus générale, il serait même avantageux de pouvoir déterminer les informations imprimées dans le rapport, ainsi que son format, en particulier l'ordre de tri lors de l'affichage : par date, par taille, par nom, par longueur du nom, par chemin, par `CRC`... Cela demande un grand remembrement du programme mais toutes les briques de base sont maintenant disponibles.

Enfin, une constatation s'impose : le calcul du `CRC` des fichiers reste parfois très long. Il pourrait être calculé préalablement, lors de l'écriture des données sur le disque, par le système d'exploitation et faire partie intégrante du système de fichiers. Ce dernier nécessiterait quelques modifications

Post-scriptum

- `gdups`, que j'ai écrit initialement en 2000, a pour fonction de rechercher l'identité de deux fichiers ou plus à l'intérieur d'une arborescence. Il ne permet pas de détecter des contenus *similaires*, juste de détecter des doublons.
- En pleine tourmente de l'affaire SCO, Eric S. Raymond a écrit un programme permettant de retrouver des zones de plusieurs lignes dont le texte est commun à plusieurs fichiers. Ce programme ainsi que sa description sont disponibles à **http://www.catb.org/~esr/comparator/comparator.html**.
- Sa particularité est que les signatures de chaque ligne de code source sont cryptographiquement adaptées pour être publiables sans risque de divulguer le texte original. L'intérêt est de pouvoir comparer publiquement des textes dont certains seraient *secrets*.
- D'autres logiciels (dont `diff`) et algorithmes existent pour trouver un ensemble optimal de similarités entre deux fichiers ou plus, les plus puissants étant basés sur les techniques de tris, d'arbres de préfixes et l'algorithme de *Burrows-Wheeler*. Ils peuvent servir aussi bien à trouver les différences minimales entre deux fichiers (pour faire des *patches*) qu'à déterminer les similarités entre plusieurs génomes.

mais gagnerait aussi au niveau de la tolérance aux erreurs matérielles et logicielles.

La signature d'un bloc envoyé sur un disque peut être calculée par une unité matérielle et je crois que cela est déjà possible avec certains *chipsets* de PC.

Il reste en fait à intégrer une méthode d'accès générique dans l'interface `POSIX` pour que `lstat()` puisse lire le `CRC` de manière transparente et portable.

A GNU touch

Plus je relis ce (vieux) programme, plus je lui trouve de défauts et la plupart des valeureux *beta-testeurs* ont fait des remarques similaires.

Tout d'abord, `gdups` est loin de respecter le ***GNU** Coding Standard*, ce qui est assez fâcheux étant donné le nom de ce programme. La première action est donc de remplacer tous les `exit(nombre)` par des `exit(EXIT_SUCCESS)` ou `exit(EXIT_FAILURE)`. Encore une bonne habitude à prendre, tout comme arrêter de mettre des `return` dans le `main()`...

Un autre souci concerne certaines limites sous forme de constantes et de formats de données. L'exemple le plus simple est la taille limitée des noms de fichiers : avec l'explosion des systèmes de partage de fichiers, euh, multimédia, 255 caractères n'est plus inenvisageable en pratique pour un nom de fichier. Qu'à cela ne tienne : de `byte`, on change la variable de taille en `short int`.

Cependant, cela alourdit la lecture du code qui devrait recourir à des *casts* partout où `struct noeud_fic` est utilisé. C'est aussi une bonne occasion de réécrire cette structure qui passe mal sur certains compilateurs.

Je propose donc de remplacer `unsigned char nom[0];` par `short int taille_nom; char nom[2];` Comme précédemment, on s'attend à ce que les limites ne soient pas vérifiées, et la taille de `nom[2];` est choisie pour que l'ensemble formé avec `taille_nom` soit aligné sur 4 octets.

Alors, le code s'éclaircit : on peut imprimer la chaîne simplement par `printf(p -> nom);`, l'*offset* a disparu avec les bugs qu'il pourrait causer. Si on avait défini `unsigned char nom;` il aurait fallu écrire `printf(&(p -> nom));`. Le "`&`" sert à préciser qu'on a bien affaire à un pointeur et son oubli causerait quelques soucis. De plus, en raison de l'alignement de la structure sur 8 octets, qu'on définisse une chaîne sur un octet ou deux revient au même en termes de place.

Enfin, la comparaison des chaînes de caractères ne change pas beaucoup, mais il faut commencer avec un index de `sizeof(taille_nom)` si la longueur doit être comparée dans la foulée.

Se débarrasser complètement des limites de longueur des chemins nécessite par contre beaucoup plus de modifications sur les structures de données et les algorithmes. Tout d'abord, il faut changer la stratégie de stockage des chemins, dont la copie intégrale était conservée à chaque fois afin que l'impression se résume à `printf("chemin : %s\n",&(t->chemin->nom[1]));` dans les versions précédentes de `gdups`. C'était facilement utilisable mais obligeait la duplication inutile du nom des répertoires parents, occupant donc plus de mémoire que nécessaire.

Pour économiser un peu de place, on va transformer la définition d'un chemin pour créer une liste chaînée:

```
struct noeud_chemin {
  dev_t dev;
  ino_t inode;
  struct noeud_chemin * parent;
  /* le nom en lui-même : */
  short int taille_nom;
  char nom[2];
}
```

Au moment de la création d'un nouveau nœud de chemin, la liste chaînée doit être remontée vers la racine pour vérifier que l'`inode` et le `device` sont uniques. Si jamais on retrouve les mêmes dans la liste chaînée, cela veut dire que l'on a détecté une *boucle de hardlink* et qu'il n'est pas nécessaire d'explorer ce répertoire, qui peut être oublié.

```
/* renvoie 1 si une boucle est détectée */
void hardlink (struct noeud_chemin * chemin) {
  struct noeud_chemin * p = chemin;
  do {
    /* suit la liste */
    p = p -> parent;
    /* fin de la liste : OK */
    if (p == NULL)
      return 0;
  } while ((chemin -> inode != p -> inode)
     && (chemin -> dev != p -> dev));
  /* si on arrive là, c'est qu'il y a une boucle */
  return 1;
}
```

Lors de l'impression du chemin, on va aussi remonter la liste chaînée et imprimer chaque élément. A un détail près : cela doit être effectué dans l'ordre inverse ! Pour des

raisons de place, il n'est pas envisageable de créer un autre pointeur dans la structure de description, il faut ruser.

Pour ne pas se dépayser, le plus simple est de créer une autre fonction récursive qui balaie la liste chaînée dans la phase d'appel, et imprime le chemin lors du retour :

```
void imprime_chemin_recursif (struct noeud_chemin * chemin) {
  if (chemin -> taille_nom != -1) {
    imprime_chemin_recursif (chemin -> parent);
    printf("%s/",chemin -> nom);
  }
}
```

On évite ainsi toute manipulation de grosses chaînes de caractères, et on reporte la limite de la profondeur d'exploration vers le contrôle de la taille de la pile (voir `ulimit` dans `man 1 bash`).

Les modifications de `gdups` ne s'arrêtent pas encore. Il faut régler le problème de la mémorisation du chemin lors de l'exploration, pour l'invocation de `lstat()`.

La première méthode consiste à gérer dynamiquement la taille de la chaîne, quitte à utiliser `realloc()` si le chemin est trop long.

En plus des considérations concernant `realloc()`, la structure du chemin a changé et empêche de recopier tout directement.

La solution retenue consiste à utiliser `chdir()` pour se placer dans le répertoire désiré avant de boucler sur `readdir()`.

Avant que `explore_recursif` ne s'appelle lui-même, il suffit d'appeler `chdir(nom_local)` puis `chdir("..")` après le retour.

Il faut aussi prévoir le cas où le programme devait s'arrêter prématurément : l'utilisateur ou le script se retrouverait dans le répertoire où s'est produite l'erreur. `atexit()` vient à la rescousse, il a juste besoin d'une fonction contenant `chdir()` avec en paramètre le chemin obtenu au départ par `getcwd()`.

C'est peut-être un peu compliqué mais on évite ainsi de gérer des longues chaînes de caractères de tailles variables.

Reste encore le problème du mécanisme des chemins interdits qui vient d'être cassé par toutes ces modifications.

Pour reconstruire ce système, on va réutiliser le nouveau système des chemins car il gère dynamiquement l'allocation de la mémoire et il contient les numéros d'`inodes` déjà balayés.

L'idée est de créer, à la racine de l'arbre, une liste de chemins contenant les `inodes` des chemins interdits : ceux-ci ne seront donc pas explorés si on les rencontre plus tard.

Par contre, il faut légèrement modifier la fonction récursive d'affichage car il imprimerait ces chemins interdits devant la racine.

La condition d'arrêt de la liste est un pointeur `NULL`, mais pour arrêter l'affichage, une taille négative du nom est choisie (il n'est d'ailleurs pas besoin de mémoriser le nom dans cette liste, ce qui économise un peu de place).

Nous arrivons donc à la structure suivante :

- `/dev` : le chemin en fin de liste, son lien "suivant" est `NULL`
- `/proc`
- `/tmp`
- `/usr/tmp` : ce sont les "répertoires interdits" définis par défaut. Pour économiser de la place, leur nom n'est pas mémorisé, seulement le couple `inode+device`. La taille du nom est à `-1`.
- `/azeerty`
- `/uiop` : ce sont les "répertoires interdits" ajoutés par l'utilisateur sur la ligne de commande, leur nom n'est pas mémorisé non plus et la taille du nom est à `-1`.
- `/home/user/whygee` : c'est le répertoire courant, où `gdups` a été lancé. C'est le chemin complet obtenu par `getcwd()`.
- `repertoire1`
- `repertoire2`
- `sous-repertoire1`
- `sous-sous-repertoire1` : ce sont les chemins explorés. Leur pointeur "`parent`" pointe vers leur répertoire parent, et leur "`adresse`" (`inode+device`) est comparée avec celle de tous les parents, y compris celles des chemins interdits.

Pour afficher le chemin, on arrête de remonter quand on atteint le répertoire racine.

Tout nouveau chemin exploré est donc ajouté en bas de la liste, et on la balaie jusqu'en haut pour détecter une boucle ou un chemin interdit.

La version actuelle de `gdups` contient encore beaucoup de changements et certaines simplifications importantes que je ne peux décrire intégralement dans cet article qui est déjà très long.

Et voilà !

Bien que destiné initialement à guider les débutants de la programmation en C sous Linux, grâce à des sources de complexité croissante, expliqués et commentés en français et écrits dans un but pédagogique, le programme est devenu un utilitaire pratique pour les administrateurs ou les utilisateurs consciencieux (ou bordéliques comme moi).

Il est librement modifiable et redistribuable, permettant d'ajouter de nombreuses options et améliorations dont certaines sont déjà nécessaires.

Les sources sont disponibles sur mon site web à **http://f-cpu.seul.org/whygee/lm-gdups**. N'hésitez pas à les examiner, car ils permettent de retracer progressivement toutes les étapes de la construction de cet utilitaire.

Il serait aussi utile de créer un paquet Debian et je cherche donc un volontaire pour en devenir le mainteneur (je ne suis pas assez débianiste, et puis le programme est assez simple pour poser peu de problèmes de maintenance).

J'avoue avoir dépassé les objectifs initiaux de faire "juste un petit programme d'exemple". Toutefois, sous GNU/Linux, avec un bon algorithme et un peu de doigté, on peut créer des programmes utiles et intéressants, à la fois pour les apprentis programmeurs et les utilisateurs. `gdups` a atteint la version 0.7 et ce n'est probablement qu'un début.