« Exact cover »

Le problème de la couverture exacte

On part d'un ensemble de *NB* nombres, et de *NBSE* parties de cet ensemble. Par exemple, on a l'ensemble {1, 2, 3, 4, 5} de *NB* = 5 nombres, et les parties

1 : {1 5}
2 : {2 4}
3 : {2 3}
4 : {3}
5 : {1 4 5}
6 : {1 3 5}
7 : {2 5}
8 : {1 4}
soit *NBSE* = 8 parties.

Il s'agit de trouver quelles parties choisir pour recouvrir exactement l'ensemble initial. Cela signifie que l'union de ces parties doit donner l'ensemble, et qu'aucune partie ne doit chevaucher une autre. Il s'agit de ce que l'on appelle une partition de l'ensemble. Dans notre exemple, une solution est 1, 2, 4, soit {1 5}{2 4}{3}. L'algorithme de l'*exact cover* consiste à trouver toutes les solutions du problème.[1] Il existe plusieurs algorithmes à ce sujet. Nous ne donnons deux ci-dessous, le premier étant plus facile à implanter, et le second beaucoup plus performant lorsque les nombres des éléments et des parties sont élevés. Dans tous les cas, le programme commence par entrer les données. En gardant notre exemple, on se donne *NB* = 5 et *NBSE* = 8, et l'on appelle la fonction *entreedesparties*(). Celle-ci remplit un tableau *se*[*NBSE*+1][*NB*+1], la partie numéro *i* est placée dans *se*[*i*][], la zone occupée ayant pour longueur *nbelemse*[*i*] :

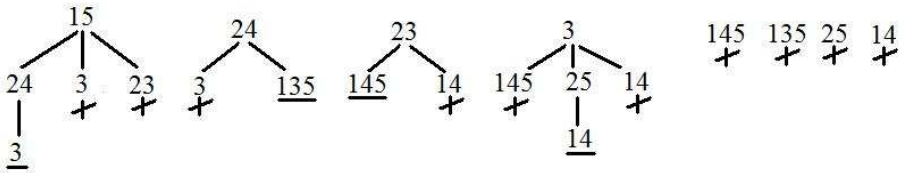
```
void entreedesparties(void)
{  se[1][1]=1; se[1][2]=5; nbelemse[1]=2;
   se[2][1]=2; se[2][2]=4; nbelemse[2]=2;
   se[3][1]=2; se[3][2]=3; nbelemse[3]=2;
   se[4][1]=3; nbelemse[4]=1;
   se[5][1]=1; se[5][2]=4; se[5][3]=5;nbelemse[5]=3;
   se[6][1]=1; se[6][2]=3; se[6][3]=5;nbelemse[6]=3;
   se[7][1]=2; se[7][2]=5; nbelemse[7]=2;
   se[8][1]=1; se[8][2]=4; nbelemse[8]=2;
}
```

Premier algorithme

L'ordre dans lequel on prend les parties qui recouvrent un ensemble est sans importance. Aussi choisit-on de prendre les parties dans l'ordre de leurs numéros

[1] Evidemment si aucune des parties ne contient un certain élément de l'ensemble, il n'y a pas de solution. Nous supposerons qu'il n'en est pas ainsi.

croissants. On commence par choisir la partie 1, qui est la racine d'une future arborescence. Puis on prend comme successeurs toutes les parties acceptables de numéro supérieur à 1. Pour chacune d'elles, on prend les parties acceptables de numéro supérieur à elle, etc. Par partie acceptable, on entend une partie dont aucun élément n'est présent parmi les prédécesseurs dans la branche concernée de l'arbre. Quand on a fini, on redémarre une arborescence à partir de la partie 2, avec ses successeurs de numéro supérieur, et ainsi de suite. Voici ce que l'on obtient dans l'exemple choisi :



On trouve ainsi les quatre solutions du problème.

Le programme principal se contente d'appeler une fonction *arbre*(*i*, 1) sur chaque partie *i* qui est mise à l'étage 1 de l'arborescence de racine *i*. Par la même occasion on met à 0 (NON) un tableau *used*[*NB*+1], indiquant qu'aucun des *NB* éléments de l'ensemble initial n'a jusqu'à présent été utilisé. Ce tableau est évidemment déclaré en global.

```
int main()
{   int i,j;
    entreedesparties();
    for(i=1;i<=NBSE; i++)
      { for(j=1;j<=NB;j++) used[j]=0;
        arbre(i,1);
      }
    getchar();return 0;
}
```

La fonction *arbre*()

La fonction *arbre*(*i, etage*) a pour arguments la partie *i* et l'étage où l'on se trouve dans l'arbre. Rappelons que l'arbre commence à l'étage 1. On commence par mettre les éléments de la partie *i* comme étant utilisés et l'on fait *used*[*se*[*i*][*j*]]=*etage* pour chaque élément numéro *j* de la partie. Remarquons que l'on met *used* à *etage*, c'est-à-dire un nombre supérieur ou égal à 1, et non pas seulement à OUI (ou 1) par opposition à NON (ou 0). Cela s'expliquera plus tard.

Le programme récursif commence ensuite par le test d'arrêt. On regarde si tous les *NB* éléments ont été utilisés lors des appels successifs de la fonction *arbre*(), et si tel est le cas, on affiche la branche de l'arborescence de la racine jusqu'à la partie i, ce qui constitue une solution du problème. Sinon, on prend tous les successeurs acceptables de la partie *i*, c'est-à-dire ceux qui n'ont pas un élément déjà utilisé précédemment, et c'est avec eux qu'on rappellera la fonction *arbre*() à l'étage suivant. Mais là un problème se pose.

Reprenons la première descente dans l'arbre dont la racine est la partie 15. Le tableau *used* devient 10001. Puis c'est la partie 24 qui est choisie en premier, le tableau *used* devient 12021. Puis la descente continue jusqu'à la partie 3, et le tableau *used* devient

12321. Tous les éléments étant utilisés, on vient de trouver la première solution, soit {15}{24}{3}. Alors se produit la première remontée dans l'arbre, là où de nouvelles parties sont en attente. Dans le cas présent, on se retrouve à l'étage 2 avec la partie 24. Il convient que *used* redevienne 12021 alors qu'il était 12321. Il suffit pour cela de mettre à 0 tous les éléments du tableau qui ont un nombre supérieur à l'étage 2 où l'on se trouve actuellement, ici le nombre 3 correspondant à l'élément 3. On retombe sur 12021. Cela explique pourquoi on a choisi de mettre *used* à *etage*. Ayant ainsi remis les pendules à l'heure, on s'aperçoit qu'à partir de la partie 24 de l'étage 2, aucune autre partie ne convient. Aucune solution n'est trouvée, et le programme remonte vers les nœuds ayant encore des branches ouvertes en descente. De là découle le programme.

```
void arbre(int i, int etage)
{ int j,k,e,allused,flag;
   for(j=1; j<=nbelemse[i]; j++) used[se[i][j]]=etage;
   allused=1;
   for(j=1;j<=NB;j++) if (used[j]==0) { allused=0; break;}
  if (allused==1)
    { number++; printf("\n%d : ",number);
      for(k=2; k<=etage ; k++)  printf("%d ",pred[k]);
      printf("%d ",i);
    }
  else
    { for(e=i+1; e<=NBSE; e++)  /** chaque partie de numéro > i */
        {  for(j=1;j<=NB;j++) if (used[j]>etage) used[j]=0;  /** actualisation */
           flag=0;                          /* si flag = 1, on ne prend pas la partie */
           for(j=1;j<=nbelemse[e];j++)   if (used[se[e][j]]!=0) { flag=1;break:}
           if (flag==0)    { pred[etage+1]=i; arbre(e,etage+1);}
        }
    }
}
```

Deuxième algorithme : les liens dansants

Les parties vont être placées dans une matrice de *NB* colonnes et *NBSE* lignes, où l'élément *m*[*l*][*c*] de la ligne *l* et de la colonne *c* est mis à 1 lorsque le nombre *c* est dans la partie *l*, et sinon il est mis à 0. Dans notre exemple, cela donne la matrice :

	1	2	3	4	5
1	1				1
2		1		1	
3		1	1		
4			1		
5	1			1	1
6	1		1		1
7		1			1
8	1			1	

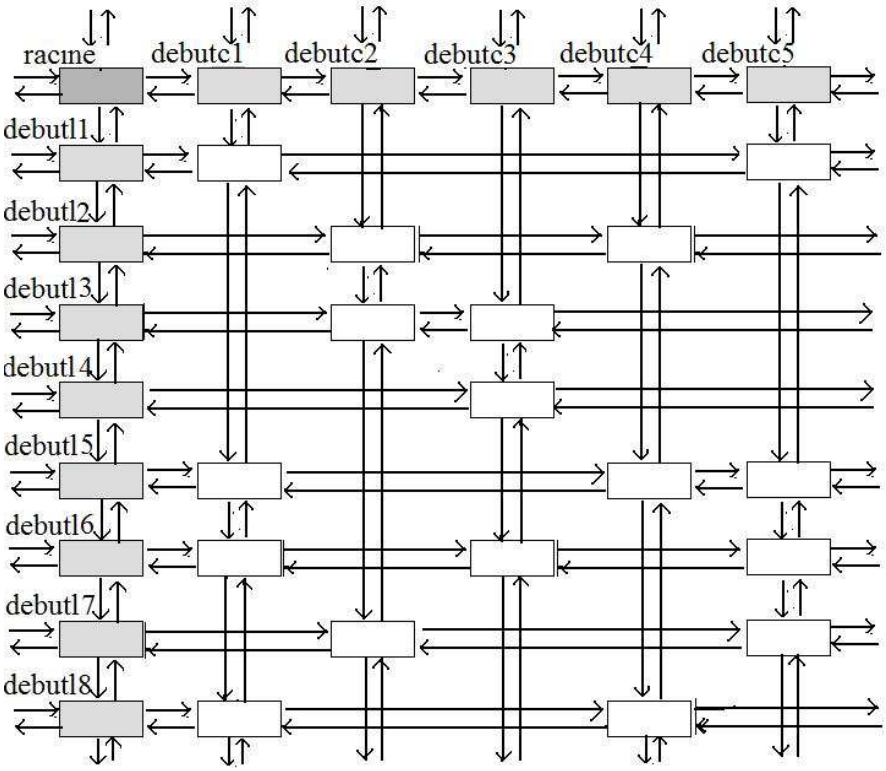
Nous n'avons pas mis les 0 qui ne jouent aucun rôle. La construction de cette matrice va utiliser des listes doublement chaînées, où seules les cellules contenant 1 vont être prises. La fonction *matrice*() va commencer par la fabriquer en utilisant des en-têtes pour chaque colonne et pour chaque ligne, puis on enlèvera les en-têtes des lignes.

Construction de la matrice par des listes doublement chaînées

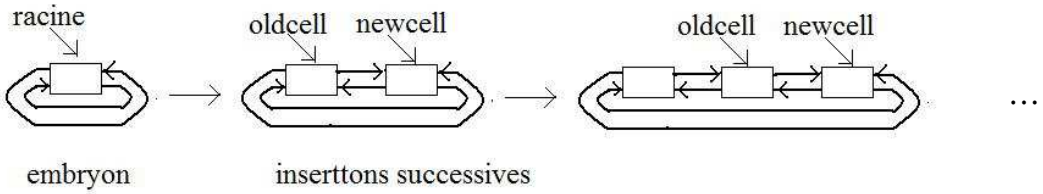
La cellule de base contient les deux nombres correspondant à la ligne et à la colonne où la cellule est placée, ainsi que quatre pointeurs vers les cellules à droite, à gauche, en haut et en bas, soit

```
struct cell
  { int l; int c;
    struct cell * d; struct cell * g; struct cell * h; struct cell * b;
  };
```

L'objectif est d'obtenir la configuration suivante des cellules, liées dans des listes doublement chaînées circulaires (la fin rejoignant le début) horizontales et verticales :



Comme toujours, la fabrication d'une liste chaînée circulaire se fait en deux temps : création de l'embryon de la liste (avec la racine, ou le début), puis insertions successives des cellules, chaque nouvelle cellule étant placée entre l'ancienne cellule - celle placée le coup d'avant, et le début de la liste.



Dans la fonction *matrice*(), on commence par créer la racine, qui donnera le moyen d'accéder à toute la matrice :

```
racine=(struct cell *) malloc(sizeof(struct cell));
racine->l=0, racine->c=0; racine->b=racine; racine->h= racine;
racine->d=racine; racine->g= racine;
```

Cette cellule racine sert d'embryon pour les deux listes circulaires des en-têtes, celle de la bordure verticale des butoirs (en-têtes) de lignes (appelés *debutligne*[]), puis celle de la bordure horizontale des butoirs de colonnes (notés *debutcolonne*[]) :

```
/** bordure gauche verticale */
   oldcell=racine;
   for(i=1;i<=NBSE; i++)
   {  debutligne[i]=(struct cell *) malloc(sizeof(struct cell));
      debutligne[i]->l=i; debutligne[i]->c=0;
      oldcell->b=debutligne[i]; racine->h=debutligne[i];
      debutligne[i]->b=racine; debutligne[i]->h=oldcell;
      oldcell=debutligne[i];
   }
   /** bordure haute horizontale */
   oldcell= racine;
   for(j=1;j<=NB; j++)
   {  debutcolonne[j]=(struct cell *) malloc(sizeof(struct cell));
      debutcolonne[j]->l=0; debutcolonne[j]->c=j;
      oldcell->d=debutcolonne[j];  racine->g=debutcolonne[j];
      debutcolonne[j]->d=racine; debutcolonne[j]->g=oldcell;
      oldcell=debutcolonne[j];
   }
```

Une fois placés les bordures des en-têtes, passons à la matrice elle-même. On va construire les listes circulaires de chaque ligne l'une après l'autre. Chaque cellule créée (*newcell*) correspond à la présence d'un 1 dans la matrice. Située dans la ligne *i* et la colonne *j*, elle correspond à l'élément numéro *j* de la partie numéro *i*, d'où *newcell->l = i* et *newcell->c = se*[*i*][*j*]. Pour la ligne i correspondant à la partie i, l'embryon de la liste est formé du butoir *debutligne*[*i*], puis on insère progressivement les cellules de la ligne, au nombre de *nbelemse*[*i*]. Chaque nouvelle cellule est placée entre la cellule placée auparavant (*oldcell*) à sa gauche et de *debutligne*[*i*] à sa droite. Mais il faut aussi la relier à ses voisines au-dessus et au dessous. La cellule est placée entre la cellule du dessus construite auparavant, pointée par *oldcellcol*[*j*] et le début de la colonne, soit *debutcolonne*[*j*]. Au départ, les pointeurs qui courent verticalement dans chaque colonne, soit *oldcellcol*[*j*] sont placés chacun en *debutcolonne*[*j*].

```
/** construction des lignes comme listes chaînées, ainsi que les colonnes */
for(j=1;j<=NB;j++) oldcelcol[j]=debutcolonne[j];
for(i=1;i<=NBSE;i++)
{ oldcell=debutligne[i];
  for(j=1;j<=nbelemse[i];j++)
   { newcell=(struct cell *) malloc(sizeof(struct cell));
     colonne=se[i][j];   newcell->l=i; newcell->c=colonne;
    /* liens droite-gauche */
     newcell->d=debutligne[i]; newcell->g=oldcell;
     oldcell->d=newcell;   debutligne[i]->g=newcell;
     oldcell=newcell;
    /* liens haut-bas */
```

```
        newcell->h=oldcelcol[colonne]; newcell->b=debutcolonne[colonne];
        oldcelcol[colonne]->b=newcell;    debutcolonne[colonne]->h=newcell;
        oldcelcol[colonne]=newcell;
       }
    }
```

Voilà, on vient de construire la matrice correspondant au dessin ci-dessus. Grâce à ses en-têtes, on peut y accéder aussi bien ligne après ligne que colonne après colonne. Notamment, si l'on veut afficher la colonne *j*, il suffit d'appeler la fonction :

```
void affichercolonne(int j)
{  struct cell * ptr;
   printf("\n%d: ",j);
   ptr=debutcolonne[j]->b;
   do {   printf("%d ",ptr->l);   m[ptr->l][j]=1;     ptr=ptr->b;   }
   while (ptr!=debutcolonne[j]);
}
```

Et pour afficher toutes les colonnes, on appelle :

```
void afficherlescolonnes(void)
{  struct cell * ptr;
   ptr=racine->d;
   while(ptr!=racine)  {  affichercolonne(ptr->c);   ptr=ptr->d; }
}
```

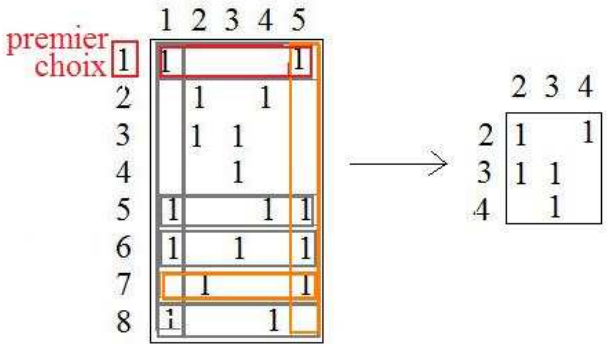
Mais pour accéder à la matrice à partir de la racine, il suffit d'utiliser les en-têtes des colonnes, et l'on n'a pas besoin des en-têtes des lignes. Détruisons finalement les butoirs des lignes : il suffit de déplacer deux liens gauche-droite, en sautant par-dessus les en-têtes de lignes. On verra plus tard l'intérêt d'avoir supprimé ces butoirs de lignes.

```
for(i=1;i<=NBSE;i++)  {(debutligne[i]->g)->d=debutligne[i]->d;
                       (debutligne[i]->d)->g=debutligne[i]->g;
                      }
```

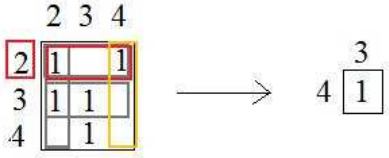
La fonction *matrice*() est terminée.

Algorithme de la recherche des solutions

Prenons l'élément 1. Grâce à la colonne 1 de la matrice, on constate qu'il est présent dans les lignes (les parties) 1, 5, 6, 8. Choisissons dans un premier temps la ligne 1, comme premier élément d'une solution potentielle du problème. On va alors supprimer la colonne 1, puisqu'elle ne sert plus. Par la même occasion, on supprime les lignes 1, 5, 6, 8, qui ne serviront plus. On constate ensuite que l'élément 5 est présent dans la ligne 1 choisie. Cela nous conduit à supprimer la colonne 5 qui ne servira plus, ainsi que la ligne 7, qui contient un 1 dans la colonne 5, et qui est la seule ainsi, autre que les lignes déjà supprimées. Après ce choix de la partie 1, il reste une matrice réduite :



Le reste de la solution, autre que la partie 1, est à trouver dans ce qui reste. Prenons l'élément 2. Il est présent dans les lignes (parties) 2 et 3. Choisissons dans un premier temps la ligne 2, comme deuxième élément de la solution éventuelle, après la partie 1. On peut maintenant supprimer la colonne 2, qui ne servira plus, ainsi que les lignes 2 et 3 qui contiennent un 1 dans la colonne 2. On constate ensuite que l'élément 4 est aussi dans la partie 2. On supprime donc la colonne 4, ainsi que les lignes ayant des 1 dans cette colonne, mais il n'en existe pas. On aboutit à une matrice réduite :



Le reste de la solution éventuelle, autre que les parties 1 et 2 déjà choisies, est à chercher dans la matrice réduite. On prend l'élément restant 3, et l'on choisit la seule ligne où il se trouve, à savoir la partie 4, mise dans la liste solution. Après avoir supprimé la colonne 3, désormais inutile, il n'y a plus rien à supprimer, et l'on tombe sur une matrice vide, sans aucune colonne. Tous les éléments ont été placés, et l'on vient d'obtenir une solution du problème, à savoir les parties 1, 2, 4 qui forment bien un recouvrement exact.

Mais ce n'est pas fini. Lors du choix de la colonne 2, on avait pris la ligne 2. On choisit maintenant la ligne 3, et l'on continue comme auparavant. Cela fait, on revient à la ligne 1, et l'on prend à tour de rôle les parties 5, 6, 8, comme on l'avait fait avec la partie 1.

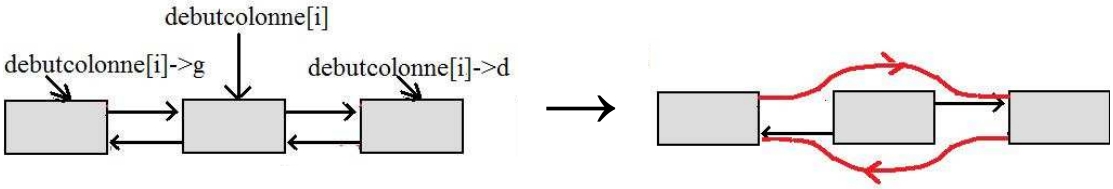
Cela va nous donner un programme récursif, où la fonction de recherche (*chercher*()) associée à une colonne se rappelle sur chaque ligne où un 1 est présent dans la colonne. D'ores et déjà, la programme principal va s'écrire ainsi :

```
int main()
{   entreedesparties();
     matrice();
    chercher();
    getchar();return 0;
}
```

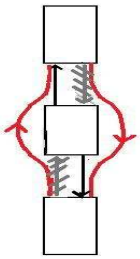
Avant d'écrire la fonction *chercher*(), on a besoin de savoir supprimer une colonne et les lignes correspondantes. C'est le rôle de la fonction *cover*(*colonne*) (ou *cacher*()). Les listes chaînées vont prendre toute leur importance, en nous facilitant la tâche.

La fonction qui cache une colonne, et son inverse

Rien de plus simple que de cacher une colonne. Il suffit de cacher son en-tête, et pour cela de déplacer deux pointeurs.

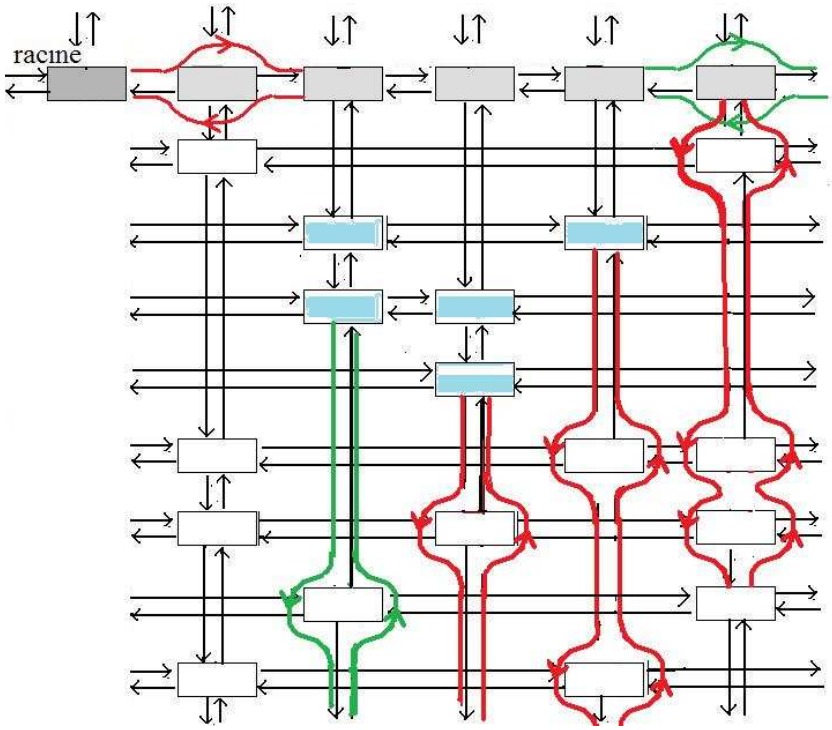


Cela fait, on ne pourra plus accéder à la colonne à partir de la racine. Tout se passe comme si elle était bel et bien supprimée. Mais remarquons que l'en-tête *debutcolonne*[*i*] est toujours là, et qu'à partir de lui on peut descendre dans la colonne. Cela étant fait, il reste à supprimer les lignes qui ont une cellule dans la colonne. Pour cela on va dans chaque ligne concernée, grâce à un pointeur courant (verticalement) dans la colonne. Chaque ligne trouvée est parcourue et ses élément supprimés. En fait on ne supprime que les éléments qui ne sont pas dans la colonne. Et là encore il suffit de déplacer deux pointeurs, verticaux cette fois.



Finalement, en accédant à la matrice via la racine, et les en-têtes de colonnes, on ne verra plus la colonne concernée et les lignes correspondantes.

Reprenons l'exemple où l'on commençait par supprimer la colonne 1 et que l'on choisissait la ligne (partie) 1. On appelle pour cela la fonction *cover*(1) puis *cover*(5) :



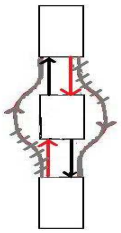
En rouge, *cover*(1), suivi de *cover*(5) en vert. Désormais les seules cellules accessibles à partir de la racine sont celles colorées en bleu

On en déduit le programme de la fonction *cover*() :

```
void  cover(int col)
{  struct cell * ptrl, * ptrc;
   (debutcolonne[col]->g)->d=debutcolonne[col]->d;
   (debutcolonne[col]->d)->g=debutcolonne[col]->g;
   ptrc=debutcolonne[col]->b;
   while(ptrc!=debutcolonne[col])
     {
        ptrl=ptrc->d;
        while(ptrl!=ptrc)
          { (ptrl->h)->b=ptrl->b;  (ptrl->b)->h=ptrl->h;
            ptrl=ptrl->d;
          }
       ptrc=ptrc->b;
     }
}
```

Lors de la recherché d'une solution, avec le rappel répété de la fonction de recherche, la fonction *cover*() est appelée à plusieurs reprises, en même temps que la matrice se réduit. Qu'on arrive ou non à une solution, il se produit ensuite une remontée dans l'arborescence récursive, et un nouveau départ à partir d'un nœud vers la recherche d'une solution. Il convient alors que la matrice retrouve la forme qu'elle avait en ce nœud. Tout ce qui avait été caché pendant la descente doit être *découvert* lors de la remontée. D'où la nécessité d'utiliser une fonction *uncover*() inverse de la fonction *cover*(). Ce qui avait été enlevé doit être remis, et pour cela on doit commencer par remettre les lignes supprimées, en allant vers le haut (en non pas vers le bas comme

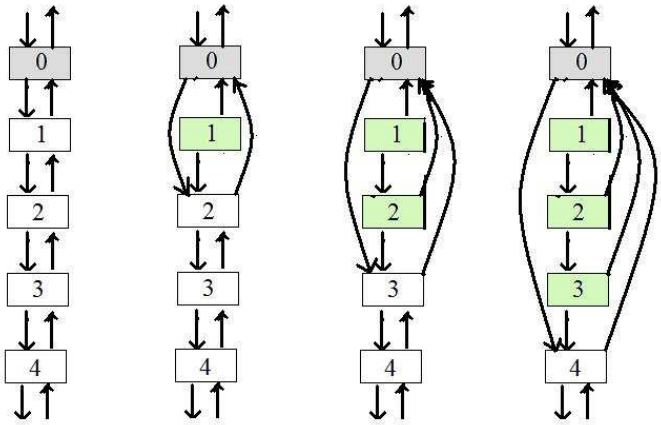
dans *cover*), puis la colonne, dans cet ordre. Ainsi, pour éviter toute embrouille, on fait les *uncover* dans l'ordre inverse des *cover*.[2]



remise en place des liens verticaux
(remplacement des deux liens en gris par ceux en rouge)

```
void  uncover(int col)
{  struct cell * ptrl, *ptrc;

   ptrc=debutcolonne[col]->h;
   while(ptrc!=debutcolonne[col])
     {
        ptrl=ptrc->g;
        while(ptrl!=ptrc)
          { (ptrl->h)->b=ptrl;  (ptrl->b)->h=ptrl;
            ptrl=ptrl->g;
          }
       ptrc=ptrc->h;
     }
   (debutcolonne[col]->g)->d=debutcolonne[col];
   (debutcolonne[col]->d)->g=debutcolonne[col];
}
```

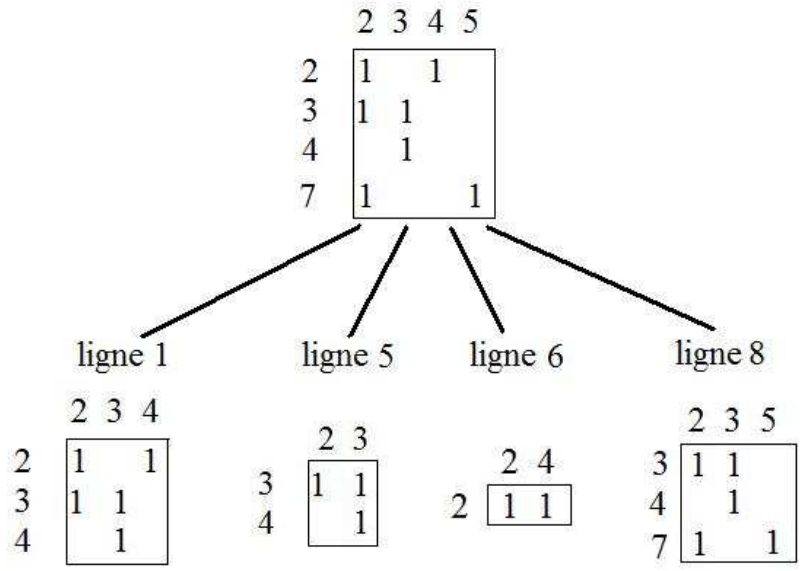


De gauche à droite, ce que produit précisément *cover*(1) sur les cellules de la colonne 5 (lignes 1,5,6,7). De droite à gauche, on a *uncover*(1).

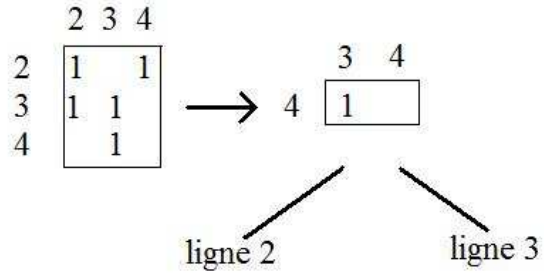
[2] Lorsque l'on prend des exemples simples, ce choix d'aller en sens inverse pour les *uncover*() ne semble pas avoir d'importance. Je n'ai pas trouvé d'exemple lumineux amenant une contradiction entre le fait d'aller dans un sens ou dans l'autre. On pourrait penser que le choix de l'ordre inverse par rapport à celui du *cover*() ne répond qu'à un réflexe de logique pure. En fait il n'en est rien dès que l'on traite des problèmes à solutions multiples, comme les pavages que l'on verra en fin de chapitre. Par exemple, si dans la fonction *chercher*() on fait un parcours de gauche à droite pour les *uncover*() au lieu de le faire de droite à gauche, le programme se bloque rapidement, annonçant une erreur fatale. Mais cela ne semble pas avoir toujours une incidence : le programme du Sudoku donné par Xi Chen (voit référence à la fin) fait des *uncover*() dans le mauvais sens, et pourtant ce programme est censé marcher.

Mise en place de la fonction de recherche

Rappelons que le programme principal appelle la fonction *chercher*(). Lors de l'appel de cette fonction, on prend la première colonne à droite de la racine, c'est la colonne 1. On va prendre l'une après l'autre toutes les parties (lignes) concernées, celles qui ont un 1 dans la colonne 1, ici les lignes 1 5 6 8. Pour chacune d'entre elles il convient de couvrir la colonne 1, ce que l'on fait donc avant de s'intéresser à chaque ligne précisément, avec la fonction *chercher*() qui va se rappeler à tour de rôle sur chacune de ces lignes.



Le programme récursif commence par prendre la ligne 1. Comme celle-ci contient aussi un 1 dans la colonne 5, on *couvre* la colonne 5 (avec les lignes correspondantes), il reste la matrice indiquée ci-dessus à gauche. On met la partie 1 dans la liste provisoire d'une solution au problème, puis on rappelle la fonction *chercher*() sur cette matrice. Pour cela on couvre la colonne 2, avec les lignes correspondantes 2 et 3, et on va rappeler la fonction sur les lignes 2 puis 3.

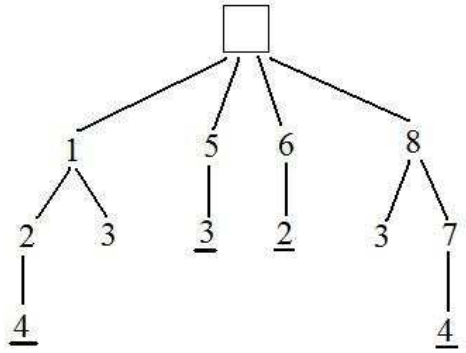


On commence par la ligne 2, on la rajoute dans la liste solution, et l'on couvre la colonne 4. Il reste la matrice

et la fonction *chercher*() est rappelée sur cette matrice. On couvre la colonne 3 (et la ligne 4). La ligne 4 est mise dans la liste solution. Il n'y a plus rien à couvrir, et l'on tombe sur la matrice vide (sans aucune colonne). Quand la fonction *chercher*() est rappelée, elle affiche la première solution trouvée 1 2 4.

Puis a lieu la première remontée, on se remet dans le contexte de la matrice obtenue avant, grâce à *uncover*(), avec la liste solution réajustée aussi. On met maintenant dans la liste solution la ligne 3 (on a gardé la ligne 1 en premier dans la liste), et l'on couvre la colonne 3, là où se trouve un 1 dans la ligne 3. Il reste une matrice réduite à la colonne 4 qui est vide. Quand la fonction *chercher*() est rappelée sur cette matrice, la colonne 4 est couverte, mais la boucle *while* ne se produit plus, puisque la colonne est vide, et la fonction *chercher*() n'est plus rappelée. Comme toutes les colonnes n'ont pas été couvertes, et qu'on ne peut pas aller plus loin, cela signifie que l'on n'a pas trouvé de solution dans ce cas.

Le processus récursif se poursuit en remontant vers un nœud ayant une branche en attente, tout en réactualisant la matrice et la liste solution.



On trouve ainsi les quatre solutions du problème.

On en déduit le programme :

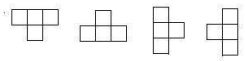
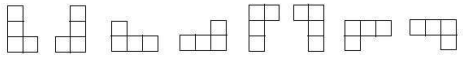
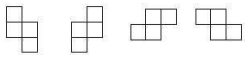
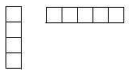
```
void chercher(void)
{    int colonne,i;
     struct cell * ptrc, *ptrl;

   if (racine->d==racine && racine->g==racine)
     { compteur++; printf(" \nsolution %d:",compteur);
       for(i=0;i<n;i++) printf(" %d ",resultat[i]);
     }
   else
     {
        colonne=(racine->d)->c;    cover(colonne);
        ptrc=debutcolonne[colonne]->b;
        while(ptrc!=debutcolonne[colonne])
          {
             resultat[n++]=ptrc->l;
             ptrl=ptrc->d;
             while(ptrl!=ptrc)  {cover(ptrl->c);    ptrl=ptrl->d;   }
             chercher();
             ptrl=ptrc->g;
             while(ptrl!=ptrc)   { uncover(ptrl->c);   ptrl=ptrl->g;  }
```

```
            ptrc=ptrc->b;
            resultat[--n]=0;
          }
        uncover(colonne);
      }
  }
```

Application : Pavages d'un rectangle par des tétraminos

Les tétraminos sont formés de quatre carrés collés face contre face. On les distingue suivant leur forme :

- forme en T 
- forme en L 
- forme en zig zag 
- carré 
- forme en I 

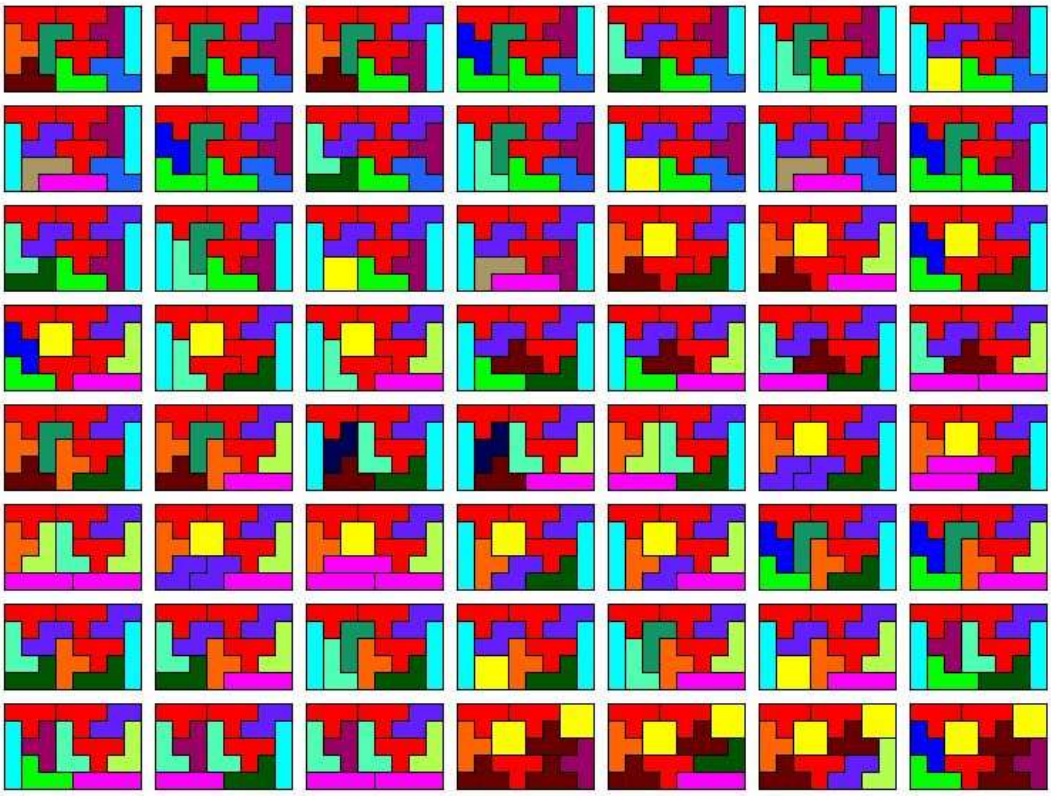
Soit au total 19 pavés possibles.

L'objectif est d'obtenir tous les pavages possibles d'un rectangle utilisant ces pavés, certains pouvant ne pas être pris, et d'autres pris plusieurs fois.

Les cases du rectangle LxH sont numérotées de 0 à $L\,H - 1$, constituant l'ensemble initial de $NB = L\,H$ nombres. On va prendre toutes les positions possibles des 19 types de pavés. Par exemple, le pavé *T* occupe les cases 0 1 2 (L+1) correspondant à la partie 0, ou 1 2 3 (L+2) correspondant à la partie 1, ou … On fait ainsi coulisser chaque type de formes dans le rectangle afin d'avoir toutes ses positions possibles. Pour $NB = L\,H = 8\,.\,5$, on arrive à 427 positions (ou parties). Il s'agit de trouver toutes les façons de choisir des parties donnant une couverture exacte. On retombe sur le problème de l'*exact cover*. C'est là qu'on mesure la différence de performance des deux algorithmes. Tant que les dimensions du rectangle sont faibles, et le nombre de pavages limité (quelques dizaines ou centaines), la différence est minime. Mais lorsque la dimension du rectangle est de 8 sur 5, avec les 800 290 solutions possibles, il faut attendre des heures avec le premier algorithme, alors que les liens dansants donnent une réponse en quelques secondes.

Voici quelques résultats donnant le nombre des pavages selon les dimensions du rectangle :

L	4	4	5	6	8	7	8	8
H	3	4	4	4	3	4	4	5
Nombre de pavages	23	117	454	2003	997	9157	40899	800290



Quelques-uns des 800 290 pavages du rectangle 8 x 5 par les tétraminos

Références bibliographiques, pour aller plus loin, notamment le traitement du jeu du Sudoku :

- D. E. Knuth, *Dancing Links*, *www-cs-faculty.stanford.edu/~uno/papers/dancing.ps.gz*
- Xi Chen, *Dancing Links*, *cgi.cse.unsw.edu.au/~xche635/dlx_sodoku/*