# *Résolution d'une structure SOMA par l'algorithme DLX de Donald Knuth.*

Par ClodoDuNet.

# Table des matières.

# Introduction.

Dans ce document, on discute des algorithmes (et de leur implémentation) utilisés pour déterminer si une figure peut être ou non construite intégralement avec les *7* pièces *A*, *B*, *L*, *P*, *T*, *V* et *Z* du jeu *SOMA*.

Le code source de tout le projet est écrit en `C++` standard; il est regroupé dans l'espace de nommage `soma`. Celui-ci possède lui-même *2* sous-espace de nommage : l'un appelé `utils`, constitué de classes et fonctions d'intérêt général (mais conçues dans l'optique du projet), et l'autre nommé `solver` qu'on va décrire dans les sections suivantes.

L'espace de nommage `solver` est lui-même subdivisé en *2* espaces de nommage. Le premier s'appelle `soma_structure`; il sera discuté dans la première partie. Il contient les fonctionnalités nécessaires pour charger les coordonnées d'une figure à résoudre et pour calculer l'ensemble des positions possibles des pièces du jeu *SOMA* dans ladite figure.

Le second sous-espace de nommage de `solver` est `dlx_algorithm` qui implémente l'algorithme *DLX* de Donald Knuth et qui sera discuté dans la partie *2*.

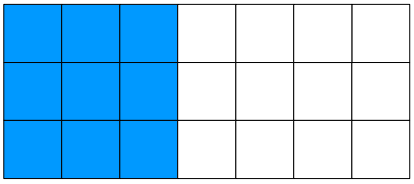
# 1) L'espace de nommage soma_structure.

Dans l'espace de nommage `soma_structure`, on trouve en premier lieu la classe `CSomaLoader`. Cette classe permet de charger une figure respectant le format de fichier décrit sur la page du défi (http://c.developpez.com/defis/5-Cube-Soma/). Pour ce faire, il suffit de spécifier le nom du fichier à son constructeur. Celui-ci lancera une exception de type `std::invalid_argument` si le fichier spécifié ne respecte pas scrupuleusement le format imposé par le règlement du jeu. Il est seulement permis de sauter des lignes entre chaque ligne de coordonnées (mais pas après la dernière!) ou de mettre des espaces après chaque coordonnée au sein d'une ligne (sauf après la dernière coordonnée!). Il est également permis d'avoir des coordonnées de cubes identiques, tant qu'il y a exactement *27* cubes aux coordonnées valides (*X*, *Y* et *Z* séparés par des virgules et compris entre *0* et *9* inclus).

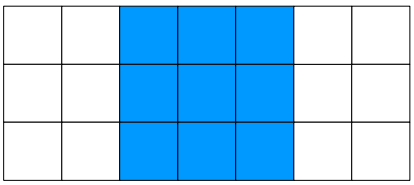
```
CSomaLoader
          (
          const char* fileName
          );
```

Chaque structure est repérée au sein de sa boîte englobante, c'est-à-dire le plus petit parallélépipède la contenant (centrée en *(0,0,0)* ainsi que la structure elle-même par conséquent). Il s'agit en fait d'un tableau de booléens à *3* dimensions; chaque élément de ce tableau vaut *1* si un cube de la structure est présent à la position de cet élément et *0* sinon. Par exemple, pour le cube de *SOMA*, la boîte englobante est évidemment le cube lui-même.

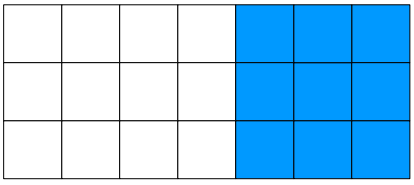
On va illustrer cette idée en construisant la boîte englobante de la structure correspondante au fichier SOMA_exemple.txt disponible sur la page du défi. On notera *E* cette figure. Pour cette figure, on constate que l'on a au maximum *7* cubes sur l'axe des abscisses, *3* sur l'axe des ordonnées et *3* également sur l'axe des *Z*, soit *7 x 3 x 3 = 63* cubes dans la boîte englobante. Le plus simple ici (pour le dessin...) est de représenter *E* dans le plan *(0XY)* (ou *(0XZ)* ) suivant les valeurs successives de *Z* (cela fait *3* couches à dessiner au lieu de *7* si on choisit le plan *(0YZ)* suivant les valeurs successives de *X*...). En bleu figurent les cubes faisant partie de *E,* (les bancs faisant partie du reste de sa boîte englobante).



Structure *E* : *Z=0*.



Structure *E* : *Z=1*.



Structure *E* : *Z=2*.

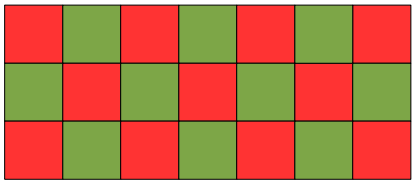
La classe `CSomaLoader` propose la fonction membre `getBoundingBox` qui permet de récupérer la boîte englobante d'une structure pour des traitements ultérieurs :

```
inline const soma::utils::general::CBitArray3D& getBoundingBox() const;
```

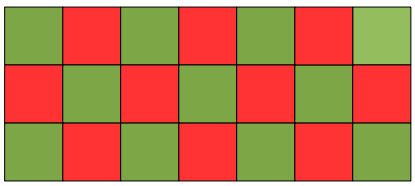
Une fois une structure chargée et avant de pouvoir la résoudre, il est nécessaire de calculer, pour chacune des pièces *A*, *B*, *L*, *P*, *T*, *V* et *Z*, l'ensemble des cubes qu'elle peut occuper dans la structure. On rappelle que les positions des cubes constituants une structure sont données relativement à sa boîte englobante et non à la structure elle-même. C'est dans la classe `CAllSomaPiecesPositions` (toujours dans l'espace de nommage `soma_structure`) que sont effectués ces calculs après avoir vérifié que les *27* cubes ont bien tous des coordonnées différentes (pas de chevauchement). Cette classe dispose d'un unique constructeur dont l'unique argument est un tableau de booléens à *3* dimensions, fourni par exemple par un objet de type `CSomaLoader` (via sa fonction membre publique `getBoundingBox`).

```
CAllSomaPiecesPositions
                        (
                        const soma::utils::general::CBitArray3D& boundingBox
                        );
```

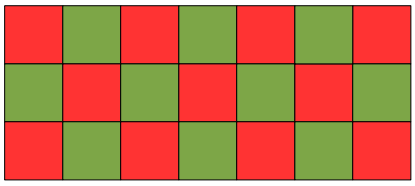
Il est possible de diminuer le nombre de positions possibles pour certaines pièces en utilisant le concept de parité. Pour comprendre ce qu'est la parité d'une structure, il faut d'abord imaginer que les cubes constituant sa boîte englobante sont colorées, alternativement de *2* couleurs différentes, disons, rouge et vert. Par exemple, la boîte englobante de la structure *E* peut être colorée de la manière suivante :



boîte englobante structure *E* colorée : *Z=0*.



boîte englobante structure *E* colorée : *Z=1*.



boîte englobante structure *E* colorée : *Z=2*.

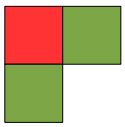
La parité d'une structure, c'est alors simplement le nombre de cubes coloriés en rouge (dans la structure évidemment) moins le nombre de cubes coloriés en verts. Ainsi, pour l'exemple de la structure *E*, la parité vaut :

*Z = 0* : *5* cubes rouges – *4* cubes verts.
*Z = 1* : *4* cubes rouges – *5* cubes verts.
*Z = 2* : *5* cubes rouges – *4* cubes verts.
Total : *14* cubes rouges – *13* cubes verts = *1*.

Le calcul de parité peut également être effectué sur les pièces du jeu *SOMA*. Seul problème, comme elles peuvent être disposées à des emplacements différents dans la structure, ce calcul donne, a priori, un résultat différent à chaque position. Cependant, on va voir qu'il y a des simplifications conduisant à des résultats intéressants.

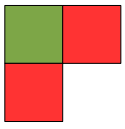
Commençons par la pièce *A*; celle-ci occupe toujours *2* couches successives de cubes d'une boîte englobante, quelque soit sa position. Voyons ce que donne un calcul de parité pour cette pièce.

Première configuration (le cube rouge de la deuxième couche est situé sur le cube vert en bas à gauche de la première couche) :

Couche *1* : 

Couche *2* : 

Deuxième configuration (le cube vert de la deuxième couche est situé sur le cube rouge en bas à gauche de la première couche) :

Couche *1* : 

Couche *2* : 

La pièce *A* peut être tournée de *12* façons différentes mais ce sont les *2* seules colorations possibles de cette pièce (avec *2* couleurs!). En effet, il est toujours possible d'observer cette pièce telle qu'elle est ci-dessus, quitte à changer le point de vue de l'observateur. La boîte englobante étant toujours bi-colorée alternativement, on revient à l'une des *2* configurations ci-dessus. Ainsi, l'on peut affirmer que la pièce *A* est toujours de parité *0* : toujours *2* cubes rouges moins *2* cubes verts.

La situation est identique pour la pièce *B* (qui est symétrique à la pièce *A*). Regardons, en effet, les *2* colorations possibles.

Première configuration (le cube rouge de la deuxième couche est situé sur le cube vert en haut à droite de la première couche) :

Couche *1* : Couche *2* :

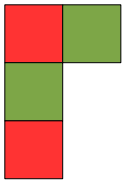
Deuxième configuration (le cube vert de la deuxième couche est situé sur le cube rouge en haut à droite de la première couche) :

Couche *1* : Couche *2* :

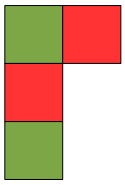
On a également toujours *2* cubes rouges moins *2* cubes verts, donc la parité de la pièce *B* vaut toujours *0*.

Examinons le cas de la pièce *L* maintenant. Contrairement aux *2* pièces précédentes, celle-ci peut toujours être visualisée sur une seule couche (moyennant des rotations du point de vue) bien qu'elle puisse être tournée de *24* façons différentes (c'est le record pour les pièces du jeu *SOMA*). Regardons les bi-colorations possibles de cette pièce.

Première configuration :



Deuxième configuration :



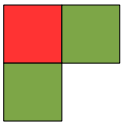
Ce sont les 2 seules colorations possibles. Une fois de plus, on toujours *2* cubes rouges moins *2* cubes verts, donc la pièce L est toujours de parité *0*.

Passons à présent à la pièce *P*. Bien que similaire aux pièces *A* et *B* (elle doit, en effet, toujours être visualisée sur *2* couches de cubes), elle ne peut être tournée que de *8* façons différentes (*12* pour les pièces A et B) et sa parité ne vaut pas *0*. Regardons, en effet, ses bi-colorations possibles.

Première configuration
(le cube vert de la deuxième couche est situé sur le cube rouge en haut à gauche de la première couche) :

Couche *2* :

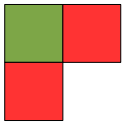
Couche *1* :



Deuxième configuration
(le cube rouge de la
deuxième couche est
situé sur le cube  vert
en haut à gauche de
la première couche) :

Couche *2* :

Couche *1* :



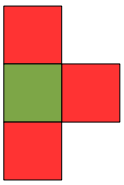
Il n'y a pas d'autre moyen de colorier la pièce *P* avec *2* couleurs. On observe ainsi *2* valeurs possibles pour la parité de *P*.

1. *1* cube rouge – *3* cubes verts = *-2* (cas correspondant à la première configuration).

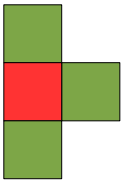
2. *3* cubes rouges – *1* cube vert = *2* (cas correspondant à la deuxième configuration).

Pour ce qui est de la pièce *T*, il est toujours possible, à l'instar de la pièce *L*, de la visualiser sur une seule couche (après rotation, si besoin, du point de vue, la pièce *T* pouvant être tournée de *12* façons différentes). Voyons quelles sont les valeurs possibles de sa parité.

Première configuration :



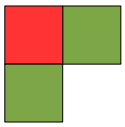
Deuxième configuration :



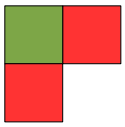
Comme la pièce *P*, la pièce *T* est donc toujours, soit de parité *2* (*3* cubes rouges moins *1* cube vert), soit de parité *-2* (*1* cube rouge moins *3* cubes verts).

Dans le cas de la pièce *V* (qui peut être tournée de *12* façons différentes), sa parité vaut *1* ou -1 comme l'indiquent ses *2* configurations possibles de *2* couleurs.

Première configuration :



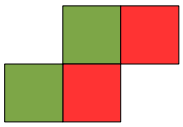
Deuxième configuration :



L'on constate ainsi que la pièce *V* possède toujours, soit *2* cubes verts et *1* cube rouge (cas de la première configuration, la parité vaut alors *-1*), soit *2* cubes rouges et un cube vert (cas de la deuxième configuration, *V* a alors pour parité *1*).

Enfin, la pièce *Z* (qui peut être tournée de *12* façons différentes), comme les pièces *A*, *B* et *L* est de parité *0*. Voyons ses bi-colorations possibles pour nous en convaincre.

Première configuration :



Deuxième configuration :

L'on a compris que cette pièce était toujours constituée de *2* cubes rouges et de *2* cubes verts, soit *2 – 2 = 0* pour la parité.

Résumons à présent les parités des différentes pièces par un tableau et essayons d'en tirer des conséquences.

| **Pièce/Parité** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *A* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *B* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *L* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *P* | *-2* | *-2* | *-2* | *-2* | *2* | *2* | *2* | *2* |
| *T* | *-2* | *-2* | *2* | *2* | *-2* | *-2* | *2* | *2* |
| *V* | *-1* | *1* | *-1* | *1* | *-1* | *1* | *-1* | *1* |
| *Z* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| Total | *-5* | *-3* | *-1* | *1* | *-1* | *1* | *3* | *5* |

Ce tableau est intéressant car il permet plusieurs remarques. Premièrement, si une structure a une solution, c'est-à-dire si elle peut être construite intégralement avec les *7* pièces du jeu *SOMA*, alors sa parité est évidemment la somme des parités des *7* pièces. D'après le tableau donc, une structure *SOMA* valide ne peut avoir comme parité que *-5*, *-3*, *-1*, *1*, *3* ou encore *5*. Autrement dit, si une structure a une parité autre que *-5*, *-3*, *-1*, *1*, *3* ou *5*, alors on est sûr qu'elle n'a pas de solution et il n'est alors pas utile de calculer les positions possibles de chaque pièce dans la structure, et encore moins de lancer quelque algorithme de résolution que ce soit.

Attention cependant! L'on a en effet ici une condition nécessaire mais pas suffisante. Par exemple, la figure en forme de *W* sur la page du défi, a pour parité *1* mais n'a pas de solution (c'est d'ailleurs ce qu'il faut démontrer...). En revanche, la figure obtenue après avoir chargé le fichier SOMA_impossible.txt (toujours sur la page du défi) a pour parité *9* et l'on sait donc d'avance qu'elle n'a pas de solution. Par conséquent, la classe `CAllSomaPiecesPositions` est dotée d'une fonction membre publique appelée `hasNoSolution` qui renvoie un booléen à vrai lorsque l'on est sûr qu'une structure n'a pas de solution. Quand elle retourne faux, on ne peut en revanche rien en déduire (solution ou pas?) et il faut donc lancer l'algorithme de résolution.

```
inline const bool hasNoSolution() const;
```

Autre remarque que l'on peut avoir après lecture du tableau des parités, c'est que l'on peut diminuer la liste des positions à analyser pour les pièces *V* (toujours), *P* et *T* (ça dépend de la parité de la structure). En effet, si une structure a une parité égale à *-5*, *-3* ou *-1*, alors la pièce *V* est de parité *-1* et l'on sait alors que les positions qu'elle occupe dans la structure et où elle est de parité *1* ne sont pas à enregistrer car elles n'aboutissent pas à une solution. Inversement, si la structure a une parité qui vaut *1*, *3* ou *5*, la parité de la pièce *V* est *1*; il ne faut donc pas tenir compte des cas où sa parité vaut *-1*.

De même pour les pièces *P* et *T*, sauf que si la structure est de parité *-1*, ou *1*, l'une est de parité *-2* et l'autre est de parité *2* (sans qu'on ne sache laquelle a priori) et on ne peut rien en déduire. En revanche, si la structure a pour parité *-5* ou *-3*, les pièces *P* et *T* ont pour parité *-2* et l'on élimine donc les positions où leur parité vaut *2*. Elles sont, inversement, toutes deux de parité *2* lorsque la structure est de parité *3* ou *5* (on élimine donc les cas où les pièces *P* et *T* sont de parité *-2*).

Les positions calculées par le constructeur d'un objet de la classe `CAllSomaPiecesPositions` sont enregistrées dans une matrice booléenne notée *M*. Soit *S* une structure à résoudre, $N_1$ le nombre de cubes sur l'axe des *X* de la boîte englobante de *S*, $N_2$ celui sur l'axe des *Y* et $N_3$ celui sur l'axe des *Z*, alors *M* possède *7* + $N_1$ *x* $N_2$ *x* $N_3$ colonnes; les *7* premières colonnes correspondant à chaque pièce du jeu *SOMA*, et donc les autres aux positions des cubes de la boîte englobante de *S*. De plus, le nombre de lignes de la matrice *M* est le nombre de façons de placer chaque pièce dans *S*.

Typiquement, une ligne de *M* dispose obligatoirement d'un unique *1* entre les colonnes *0* et *6* (incluses), identifiant une des pièces *A*, *B*, *L*, *P*, *T*, *V* ou *Z*. *3* (pour la pièce *V*) ou *4* (pour les autres pièces) autres *1*, donnant la position de la pièce dans la boîte englobante de *S*, occupent d'autres colonnes de ladite ligne. Toutes les autres colonnes de cette ligne sont occupées par des *0*. En général, il y aura des colonnes complètes de *0* car, hormis le cube de *SOMA*, aucune structure n'occupe entièrement sa boîte englobante. Cela ne gène en rien pour appliquer l'algorithme *DLX* comme on va le voir dans la seconde partie de ce document.

Pour récupérer la valeur de *M*, on peut faire appel à la fonction membre publique `getAllPiecesPositions` de la classe `CAllSomaPiecesPositions`. Le type de la matrice *M* est un `std::vector<std::vector<bool> >`.

```
inline const std::vector<std::vector<bool> >& getAllPiecesPositions() const;
```

# 2) L'espace de nommage dlx_algorithm.

L'algorithme *DLX* a été inventé par le célèbre informaticien Donald Knuth pour résoudre le problème de la couverture exacte qui est l'un *21* problèmes NP-complets de Karp. L'implémentation de cet algorithme réalisée pour résoudre une structure *SOMA* se trouve dans l'espace de nommage `dlx_algorithm`.

Pour représenter un problème de couverture exacte, on utilise une matrice booléenne (composée uniquement de *0* et de *1*). Chaque colonne de la matrice représente une contrainte spécifique et chaque ligne un moyen de satisfaire certaines des contraintes. Si on note m et n le nombre de lignes et de colonnes respectivement, alors la matrice d'un problème de couverture exacte a la forme générale suivante :

$$\begin{matrix} a_{0\,0} & a_{0\,1} & a_{0\,2} & \cdots & a_{0\,(n-1)} \\ a_{1\,0} & a_{1\,1} & a_{1\,2} & \cdots & a_{1\,(n-1)} \\ a_{2\,0} & a_{2\,1} & a_{2\,2} & \cdots & a_{2\,(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)\,0} & a_{(m-1)\,1} & a_{(m-1)\,2} & \cdots & a_{(m-1)\,(n-1)} \end{matrix}$$

où $\forall (i,\,j) \in \{0,\cdots,m-1\} \times \{0,\cdots,n-1\}, a_{i,\,j} \in \{0,1\}$

Un sous-ensemble des lignes de la matrice, noté *U*, est alors une couverture exacte si et seulement si (condition et suffisante) :

1. Certaines colonnes, appelées colonnes principales,formant un ensemble non vide, noté $C_p$ possèdent exactement un élément égal à *1*.

2. Un second ensemble, noté $C_s$ , complémentaire de $C_p$ (c'est-à-dire $C_p \cup C_s = U$ et $C_p \cap C_s = \emptyset$ ) et éventuellement vide, est constitué de colonnes, dites secondaires, possédant, au plus, un élément égal à *1*.

C'est une généralisation de la formulation originelle du problème de couverture exacte où toutes les colonnes de *U* doivent avoir exactement un élément égal à *1*. Cette généralisation est pratique pour le problème des structures *SOMA*, car, comme on l'a vu à la fin de la première partie, la matrice obtenue après calcul de toutes les positions des *7* pièces du jeu *SOMA*, peut comporter des colonnes remplies de *0*. Fort heureusement, cela ne change rien à l'algorithme *DLX* lui-même si l'on prend garde à ce qu'il ne prenne pas en compte ce type de colonnes.

Pour fixer les idées, rien ne vaut un exemple concret illustrant le problème de la couverture exacte. Voici tout d'abord une matrice booléenne à résoudre :

$$A= \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

Ce problème de couverture exacte a *3* solutions : $\{ligne_{0,}ligne_{3,}ligne_{4}\}$ , $\{ligne_{1,}ligne_{2}\}$ et $\{ligne_{2,}ligne_{4,}ligne_{5}\}$ .

L'algorithme *DLX* de Donald Knuth est en fait une implémentation particulière d'un autre algorithme, l'algorithme *X* (qu'il a également inventé d'ailleurs). Ce dernier est un algorithme de force brute, récursif, non déterministe et de parcours en profondeur (*backtracking*). Il est, de plus, capable de trouver toutes les solutions d'un problème de couverture exacte, qu'il fournit de manière analogue à l'exemple que l'on vient de donner. Voici le corps de l'algorithme *X* en pseudo-code :

*Début Algorithme X :*
*Si la matrice à résoudre, notée A, est vide alors :*
*le problème est résolue;*
*enregistrer la solution;*
*Fin si.*
*Sinon :*
*choisir une colonne C (monsieur Knuth recommande de choisir celle qui compte le moins de 1 pour minimiser le nombre de branchements);*
*choisir une ligne R telle que* $A_{R,C}=1$ *;*
*inclure R dans la solution partielle;*
*Pour chaque colonne j telle que* $A_{R,j}=1$ *faire :*
*Pour chaque ligne i telle que* $A_{i,j}=1$ *faire :*
*effacer la ligne i de A;*
*Fin pour.*
*effacer la colonne j de A;*
*Fin pour.*
*Fin sinon.*
*répéter cet algorithme récursivement sur la matrice A réduite;*
*Fin algorithme X.*

Notons que le caractère non déterministe de l'algorithme *X*, est dû au fait que celui-ci se subdivise en sous-algorithmes indépendants après qu'on ait choisi une colonne *C* (de manière déterministe). En effet, à ce stade, chaque sous-algorithme va travailler sur la même matrice réduite mais avec une valeur de *R* différente et va réduire ladite matrice en fonction de cette ligne *R* particulière. Notons également, qu'à un moment donné, si un sous-algorithme rencontre une colonne sans aucun *1*, il ne donnera pas lui-même naissance à des sous-algorithmes, il aura échoué et sa branche de récursion ne conduira à aucune solution. Évidemment, si tous les sous-algorithmes échouent, le problème de couverture exacte n'a pas de solution.

En l'état, le problème majeur de l'algorithme *X* est que la recherche de *1* sur une ligne ou une colonne peut prendre beaucoup de temps sur de grosses matrices car ces dernières sont très creuses ce qui implique que le nombre de *0* est très grand devant celui de *1*. La technique proposée par Donald Knuth, pour contourner ce problème, est de stocker uniquement les 1 des matrices sous forme de plusieurs listes doublement chaînées. De plus, l'algorithme *X* suppose que l'état de la matrice de travail soit constamment sauvegardé ou restauré, notamment durant les *backtrackings*, puisqu'il faut alors réintroduire les lignes et les colonnes supprimées.

Partant de ces observations, monsieur Knuth a mis au point la technique dite des *« listes dansantes »* (*« Dancing links »* en anglais; d'où le nom d'algorithme *DLX*, initiales de dancing et de links, suffixées du *X* de l'algorithme *X*). Cette technique, très puissante, permet d'inverser n'importe quelle opération effectuée sur une liste doublement chaînée. En effet, si *n* est un nœud d'une telle liste, alors les *2* instructions suivantes supprimeront *n* de la liste :

```
n->right->left=n->left;
n->left->right=n->right;
```

L'astuce de Donald Knuth est de ne pas détruire le nœud *n* en tant qu'objet (appel à **`delete`**, garbage collector...), pour pouvoir le réintroduire dans la liste lorsqu'il y aura besoin de restaurer l'état précédent de la matrice. Cette opération de réinsertion est réalisée via les *2* instructions suivantes :

```
n->right->left=n;
n->left->right=n;
```

Grâce à la technique des *« Dancing Links»*, on peut, sans problèmes stocker la matrice creuse sous la forme d'une structure toroïdale à base de listes doublement chaînées circulaires, dans laquelle chaque nœud modélise un *1*. Ainsi, à chaque *1* d'une ligne, on associe un nœud *w* de la structure. Celui-ci a un lien vers le précédent nœud sur la même ligne (c'est `w->left`) ainsi que vers le suivant (il s'agit de `w->right`). Mais *w* possède également un lien vers le précédent nœud sur sa colonne (`w->up`) et vers le suivant (c'est `w->down`).

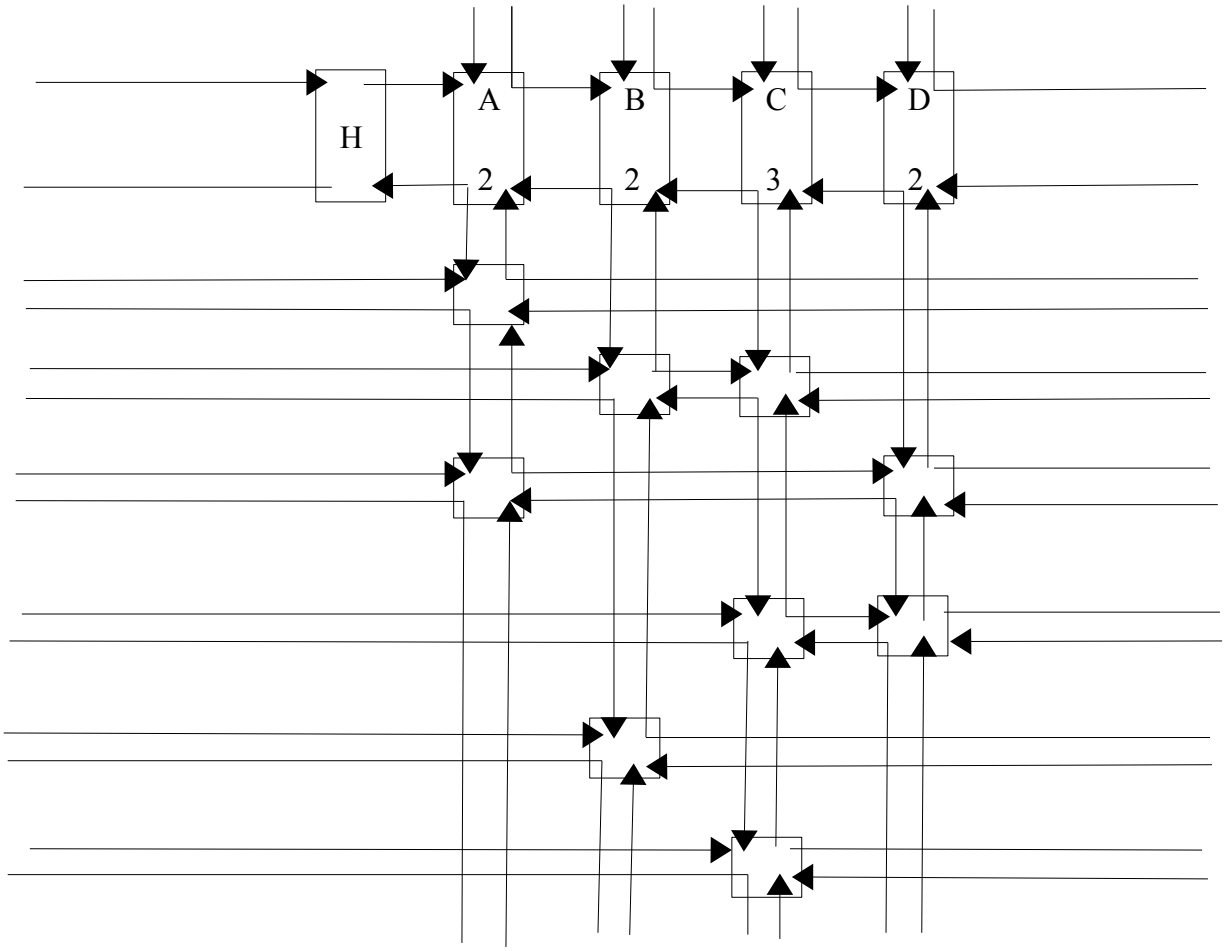
*w*, à l'instar des autres nœuds de sa colonne, pointe également vers un élément spécial, l'en-tête de sa colonne (`w->header`), lequel renseigne le nombre de *1* sur sa colonne (champs `size`), son nom (pour afficher une solution, champs `name`), ainsi que, dans notre implémentation une information sur l'indice de la colonne (également pour afficher une solution. C'est le champs `column` qui est assigné à *j* si $j<7$ , et à j-7 sinon; avec *j* l'indice de la colonne). Enfin, *w* possède un champs `row`, indiquant la ligne où se trouve le *1* qu'il modélise (c'est une fois de plus spécifique à notre implémentation, dans l'optique de simplifier l'écriture d'une solution). Pour finir, un en-tête spécial, noté *H*, est la racine de la structure.

Les listes de la structure sont circulaires : le champs `left` du premier nœud d'une ligne pointe sur le dernier nœud et le champs `right` du dernier pointe sur le premier. De même, le champs `right` de *H* pointe vers le premier en-tête (dont le champs `left` pointe à son tour vers *H*) et son champs `left` est lié au dernier en-tête (dont le champs `right` pointe à son tour vers *H*). Pour finir, le champs `down` de chaque en-tête pointe vers le premier nœud de sa colonne (dont le champs `up` pointe vers l'en-tête) et son champs `up` est lié au dernier (dont le champs `down` est lié à l'en-tête).

Illustrons ce que nous venons de voir en convertissant l'exemple de la matrice A de toute à l'heure structure toroïdale :

$$A = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

Structure obtenue après conversion de *A* :



Dans l'espace de nommage `dlx_algorithm`, c'est au constructeur de la classe `CDLXSomaSolver` qu'est dévolue la tâche de générer ce type de matrice creuse. Pour ce faire, il attend, en premier argument, une matrice booléenne pleine, qui peut par exemple être générée par un objet de la classe `CAllSomaPiecesPositions`, et lui être soumise via sa fonction `getAllPiecesPositions`. Elle ne doit pas être vide (au moins *1* ligne), compter au moins *34* colonnes (*7* pour les pièces du jeu *SOMA* plus *27* pour le nombre de de cubes de la structure à résoudre). De plus, pour chaque ligne, il doit y avoir exactement un unique élément non nul entre les colonnes *0* et *6* (incluses). Si une de ces *3* conditions n'est pas remplie, le constructeur lance une exception de type `std::invalid_agument`.

Le constructeur de la classe `CDlxSomaSolver` dispose, en outre, d'un second argument (optionnel) indiquant le nombre maximal de solutions à calculer, dans le cas où on ne souhaite pas voir l'algorithme *DLX* les calculer toutes.

```
template
 <
 typename DataT
 >
CDLXSomaSolver
            (
            const DataT& data,
            const unsigned int maxNbSolutions
            );
```

Avec les *« listes dansantes »* et la matrice booléenne mise sous forme de structure toroïdale, on dispose à présent de toutes les briques pour réécrire l'algorithme *X* en algorithme *DLX*. Ce dernier fait maintenant appel à *3* fonctions annexes, que monsieur Knuth appelle *chooseColumn*, *cover* et *uncover*.

La première, *chooseColumn*, permet de sélectionner, de manière déterministe, le premier en-tête (*header*) de colonne, noté *Col*, de la structure toroïdale que l'algorithme *DLX* va traiter. On peut, simplement, assigner `H->right` à *Col*, par exemple. Cependant, si l'on désire minimiser le nombre de branchements, il faut rechercher l'en-tête *Col* qui liste le moins de nœuds (ce qui est équivalent à choisir la colonne comptant le moins de *1* dans l'algorithme *X* originel). La fonction *chooseColumn* a alors le pseudo-code suivant :

*Début fonction chooseColumn :*
*soit s entier assigné à l'entier maximum du système;*
*soit Col de type en-tête (header);*
*Pour chaque j <-* `H->right, H->right->right` *... tant que* $j \neq H$ *faire :*
*Si* `j->size` *< s alors :*
*Col <- j;*
*s <-* `j->size;`
*Fin si.*
*Fin pour.*
*retourner Col;*
*Fin fonction chooseColumn.*

C'est cette version de la fonction *chooseColumn* qui a été retenue dans notre implémentation de l'algorithme; elle est d'ailleurs quasiment retranscrite telle quelle dans la classe `CDLXSomaSolver`. En revanche, elle est déclarée privée car il ne semble y avoir aucune raison de l'exposer dans l'interface publique de la classe.

```
inline header_t* chooseColumn();
```

Les fonctions *cover* et *uncover* sont les vraies nouveautés de l'algorithme *DLX* par rapport à l'algorithme *X*. Toutes *2* prennent comme argument un objet de type en-tête (*header* de la structure toroïdale), que l'on notera, à nouveau *Col*.

*cover* a plus précisément pour rôle d'éliminer tous les nœuds situés en-dessous de *Col*, ainsi que tous les nœuds des autres en-tête liés horizontalement à ceux de *Col*. Ces opérations reviennent, dans l'algorithme *X* originel, à éliminer les lignes qui font conflit pour une colonne donnée ainsi que la colonne elle-même. En pseudo-code, *cover* aurait le corps suivant :

*Début fonction cover :*
*Argument : Col de type en-tête (header).*
`Col->right->left` <- `Col->left`;
`Col->left->right` <- `Col->right`;
*Pour chaque i <-* `Col->down`, `Col->down->down` *... tant que* $i \neq Col$ *faire :*
*Pour chaque j <-* `i->right`, `i->right->right` *... tant que* $j \neq i$ *faire :*
`j->down->up` <- `j->up`;
`j->up->down` <- `j->down`;
`j->header->size` <- `j->header->size - 1`;
*Fin pour.*
*Fin pour.*
*Fin fonction cover.*

La fonction *uncover* réalise l'opération inverse de *cover* en rétablissant l'état précédent de la matrice. Elle traverse un en-tête *Col* vers le haut puis vers la gauche, contrairement à *cover* qui parcourt *Col* vers le bas puis vers la droite. *uncover* réintroduit ainsi les nœuds que *cover* a retirés, à la place exacte qu'ils occupaient au sein de la structure toroïdale. En pseudo-code, *uncover* s'écrit comme suit :

*Début fonction uncover :*
*Argument : Col de type en-tête (header).*
*Pour chaque i <-* `Col->up`, `Col->up->up` *... tant que* $i \neq Col$ *faire :*
*Pour chaque j <-* `i->left`, `i->left->left` *... tant que* $j \neq i$ *faire :*
`j->header->size` <- `j->header->size + 1`;
`j->down->up` <- `j`;
`j->up->down` <- `j`;
*Fin pour.*
*Fin pour.*
`Col->right->left` <- `Col`;
`Col->left->right` <- `Col`;
*Fin fonction uncover.*

On retrouve les fonctions *cover* et *uncover* quasiment à l'identique dans la classe `CDLXSomaSolver`. Mais à l'instar de *chooseColumn*, ce sont des fonctions membres privées; il n'y a, en effet, pas vraiment de raison de les rendre publiques étant donné que seul l'algorithme *DLX* en fait appel.

```
inline void cover
                (
                header_t* const col
                );


inline void uncover
                 (
                 header_t* const col
                 );
```

Grâce à *chooseColumn*, *cover* et *uncover*, le pseudo-code de l'algorithme *X* se réécrit en algorithme *DLX* de la manière suivante (fonction *search*) :

*Début fonction search :*
*Si* `H->right = H` *alors :*
*enregistrer la solution;*
*sortir de la fonction search;*
*Fin si.*
*soit Col de type en-tête;*
*Col <- chooseColumn;*
*Si* `Col->size` *= 0 alors :*
*cette branche de récursion ne conduit pas à une solution du problème;*
*sortir de la fonction search;*
*Fin si.*
*cover(Col);*
*Pour chaque r <-* `Col->down`, `Col->down->down` *... tant que* $r \neq Col$ *faire :*
*ajouter r à la liste solution partielle O;*
*Pour chaque j <-* `r->right`, `r->right->right` *... tant que* $j \neq r$ *faire :*
*cover(*`j->header`*);*
*Fin pour.*
*appel récursif de search();*
*retirer r de O;*
*Pour chaque j <-* `r->left`, `r->left->left` *... tant que* $j \neq r$ *faire :*
*uncover(*`j->header`*);*
*Fin pour.*
*uncover(Col);*
*Fin fonction search.*

Dans la classe `CDLXSomaSolver`, la fonction membre, publique, qui permet de lancer l'algorithme *DLX* s'appelle également *search* et reproduit fidèlement le pseudo-code ci-dessus.

```
inline void search();
```

Pour savoir si la fonction `search` a réussi à trouver une solution, et donc si la structure à résoudre peut être construite avec les *7* pièces du jeu *SOMA*, la classe `CDLXSomaSolver` dispose d'une fonction membre publique `hasSolution`. Celle-ci retourne un booléen à *Vrai* (**`true`**) si l'algorithme *DLX* a pu trouver une solution, et à *Faux* (**`false`**) sinon.

```
inline const bool hasSolution() const;
```

Une fois que l'on sait que `search` a convergé, après appel à `hasSolution`, on peut connaître le nombre exact de solutions qu'elle a trouvées via la fonction membre publique `getNumberOfFoundSolutions` de la classe `CDLXSomaSolver`.

```
inline const unsigned int getNumberOfFoundSolutions() const;
```

Pour récupérer une solution éventuelle calculée par l'algorithme *DLX*, la classe `CDLXSomaSolver` met à disposition la fonction membre publique `getIthSolution`. Son unique argument est le numéro de la solution à laquelle on souhaite accéder et doit être compris entre *0* et `getNumberOfFoundSolutions() -1` (inclus).

```
inline const std::map<char,std::vector<unsigned int> > getIthSolution
                                                                (
                                                                const unsigned int solutionIndex
                                                                ) const;
```

La solution prend alors la forme d'un conteneur associatif `std::map<char,std::vector<unsigned int> >` où les clefs, de type **`char`**, correspondent aux noms des pièces du jeu *SOMA* (*A*, *B*, *L*, *P*, *T*, *V* et *Z*), et les valeurs, de `type std::vector<unsigned int>`, représentent les listes des cubes occupées par ces pièces dans la boîte englobante d'une structure. Ainsi si *m* est une instance de `std::map<char,std::vector<unsigned int> >`, une instruction telle que `m[A]` permet d'accéder aux positions des cubes constitutifs de *A*.

Le tableau suivant illustre une telle construction avec l'une des solutions du cube de *SOMA* :

| **Clefs (type `char`)** | **Liste des positions des cubes (type `std::vector<unsigned int>`)** |
|---|---|
| *A* | *{0, 1, 9, 12}* |
| *B* | *{3, 4, 6, 15}* |
| *L* | *{18, 21, 24, 25}* |
| *P* | *{5, 13, 14, 17}* |
| *T* | *{20, 22, 23, 26}* |
| *V* | *{7, 8, 16}* |
| *Z* | *{2, 10, 11, 19}* |

# Conclusion.

Dans ce document, nous avions pour objectif de trouver un moyen de déterminer si une certaine structure *S*, constituée de *27* cubes à des positions différentes, peut être construite avec les *7* pièces *A*, *B*, *L*, *P*, *T*, *V* et *Z* du jeu *SOMA*.

Pour ce faire, nous avons dû trouver toutes les positions possibles de chacune des pièces dans *S*. Le concept de parité, que nous avons expliqué, nous a aidé à diminuer la taille des listes des positions pour les pièces *V* (pour laquelle c'est toujours possible), *P* et *T* (quand c'est possible, suivant la valeur de la parité de *S*).

Les positions des *7* pièces calculées et arrangées en une matrice booléenne pour former un problème de couverture exacte, nous avons généré une structure toroïdale à base de listes doublement chaînées circulaires afin d'appliquer l'algorithme *DLX* de Donald Knuth. Celui-ci nous a permis de trouver toutes les solutions éventuelles de *S*.

L'algorithme *DLX*, en plus d'être plutôt élégant (c'est une question de goût...), est, de plus, très efficace. En effet, pour le problème classique du cube de *SOMA*, il trouve les *11520* solutions en moins d'une demi-seconde, le tout sur mon vieil *Athlon XP 2600+* (sous-cadencé qui plus est...). Bravo monsieur Knuth!...

Un membre de la communauté developpez.com, dont le pseudo est monnomamoi, a fourni une archive contenant les coordonnées de *24* des figures que l'on peut voir sur la page du défi. Le tableau suivant donne le nombre de solutions que notre programme, implémentant l'algorithme *DLX*, trouve pour chacun de ces fichiers (sauf les fichiers 015.txt, 022.txt et 24.txt qui n'ont pas 27 coordonnées!), ainsi que les fichiers SOMA_exemple.txt et SOMA_impossible.txt.

| **Nom du fichier** | **Nombres de solutions** |
|---|---|
| 001.txt | *832* |
| 002.txt | *294* |
| 003.txt (même figure que SOMA_exemple.txt) | *28* |
| 004.txt | *92* |
| 005.txt | *488* |
| 006.txt | *104* |
| 007.txt | *48* |
| 008.txt | *132* |
| 009.txt | *52* |
| 010.txt | *1520* |
| 011.txt | *14* |
| 012.txt | *32* |
| 013.txt | *598* |
| 014.txt | *2346* |
| 016.txt | *328* |
| 017.txt | *64* |
| 018.txt | *4* |
| 019.txt | *2* |
| 020.txt | *56* |
| 021.txt | *41* |
| 023.txt | *32* |
| SOMA_impossible.txt | *0* |
| Cube.txt (cube de SOMA) | *11520* |
| W.txt (figure du défi à démontrer sans solution) | *0* |

# Webographie.


http://www.fam-bundgaard.dk/SOMA/SOLVER/SOLVMANU.HTM

http://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X

http://en.wikipedia.org/wiki/Dancing_Links

http://www-cs-faculty.stanford.edu/~knuth/preprints.html

http://cpp.developpez.com/faq/cpp/?filtre=OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO

http://cpp.developpez.com/faq/cpp/?page=strings#STRINGS_convert_to

http://cpp.developpez.com/faq/cpp/?page=fichiers#FICHIERS_existence

http://c.developpez.com/defis/5-Cube-Soma/

http://en.wikipedia.org/wiki/Row-major_order