Liens dansants

1 Préliminaires

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `dlx` contenant une source OCaml :

```
curl cdn.sci-phy.org/mp2i/tp17-dlx.tgz | tar xvz
```

On y trouvera un fichier source `dlx.c` à compléter, un makefile, et divers fichiers contenant des problèmes algorithmiques qui nous serviront d'exemples.

1.2 Présentation du problème

On s'intéresse aujourd'hui au problème de la *couverture exacte*. On peut le décrire de la façon suivante : étant donné une matrice M contenant des 0 et des 1, quels sous-ensemble de lignes contiennent un 1 et un seul par colonne ? Par exemple, le problème suivant :

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

a deux solutions : l'ensemble de lignes $\{0,2,4\}$, et l'ensemble $\{2,3\}$.

Ce genre de problème est très difficile à résoudre efficacement, même si le principe de retour sur trace permettra évidemment de lister toutes les solutions. Le but de cette séance de travaux pratiques est de présenter l'algorithme des *liens dansants* proposé par D. Knuth en 2000, utilisant des listes doublement chaînées pour faciliter et accélérer cette approche de retour sur trace.

1.3 Description du problème par un fichier

Afin de faciliter la description d'un problème, nous allons le décrire dans un fichier. On pourrait simplement y inscrire la matrice M, mais celle-ci contient fréquemment une grande quantité de zéros, ce serait donc du gâchis. Nous allons donc procéder autrement.

On donne un nom à chacune des colonnes de la matrice M (on utilisera typiquement une série de lettres et de chiffres). Par exemple, pour notre exemple, appelons-les simplement `A`, `B`, `C`, `D`, `E` et `F`. Ces noms sont les « *objets* » que l'on cherche à obtenir.

Chaque ligne de la matrice M correspond à une *offre*. On va simplement lister, pour chaque ligne de M, les objets que ladite offre propose de fournir.

Dans le fichier, on commencera par une ligne listant les noms de tous les objets (séparés par des espaces), puis, une série de lignes contenant les offres, avec pour chacune les objets qu'elle peut fournir. Le problème initial sera donc décrit par exemple par le fichier suivant (on utilisera l'extension « `.dlx` » pour les fichiers contenant de tels problèmes) :

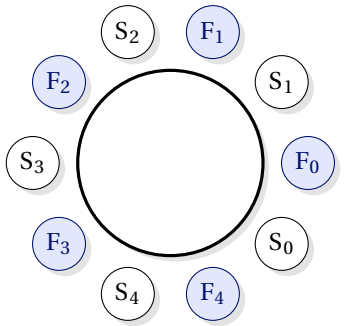
```
A B C D E F
A C
B E
B E F
A C D
D
```

Par simplicité, on n'exigera pas que les objets dans chaque offre soient listés dans le même ordre que dans la liste des objets fournie dans la première ligne, mais on supposera que chaque objet apparaît au plus une fois sur chaque ligne.

1.4 Placement à table

Cette façon de présenter les choses permet de décrire et de résoudre de très nombreux problèmes. Considérons par exemple la question du placement de cinq couples mixtes [1] autour d'une table ronde. On veut alterner hommes et femmes, et on souhaite éviter que chaque personne ne soit pas assise à côté de son partenaire.

On note $F_0, F_1, \ldots F_4$ les femmes et $M_0, M_1, \ldots, M_4$ les hommes. On peut supposer sans perte de généralité que les femmes ont déjà été placées autour de la table, et on cherche les placements possible des hommes. Notons $S_0, S_1, \ldots, S_4$ les placements possibles. On a donc ce genre de situation :



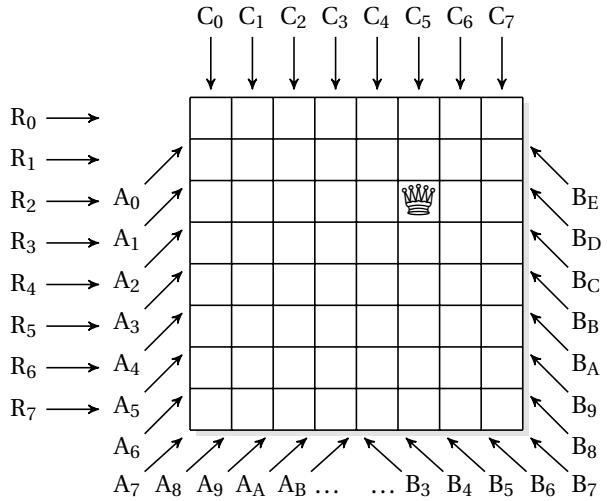
1. Il s'agit simplement ici de formaliser un problème algorithmique !

À la place S_0, on peut placer M_0, M_1 ou M_2. De même pour les quatre autres places. Le but est d'associer deux à deux chacun des cinq sièges et les cinq hommes à placer. Associer la place S_0 à l'homme M_2 peut être décrit par l'objet (ici une proposition de placement) « `S0 M2` ». On souhaite alors trouver un ensemble de propositions de placement où chaque siège trouve preneur, chaque homme est placé, et où chaque siège ne sert qu'une seule fois et chaque homme n'est assis qu'à un endroit! Il s'agit donc naturellement d'un problème de couverture exacte, et le fichier correspondant au problème est décrit dans le fichier `diner.dlx`.

1.5 Problème des huit reines

Là aussi, le problème peut être décrit en terme de couverture exacte. Deux reines ne devant pas se menacer mutuellement, le placement d'une reine requiert la « réservation » d'une ligne (« objets » `R0` à `R7`), d'une colonne (`C0` à `C7`) et de deux « diagonales » (`A0` à `AE`, comme illustré ci-dessous en utilisant une numérotation en hexadécimal, et `B0` à `BE`), à son usage exclusif.

Par exemple, la reine ci-dessous réserve la ligne `R2`, la colonne `C5` et les diagonales `A7` et `BA` :



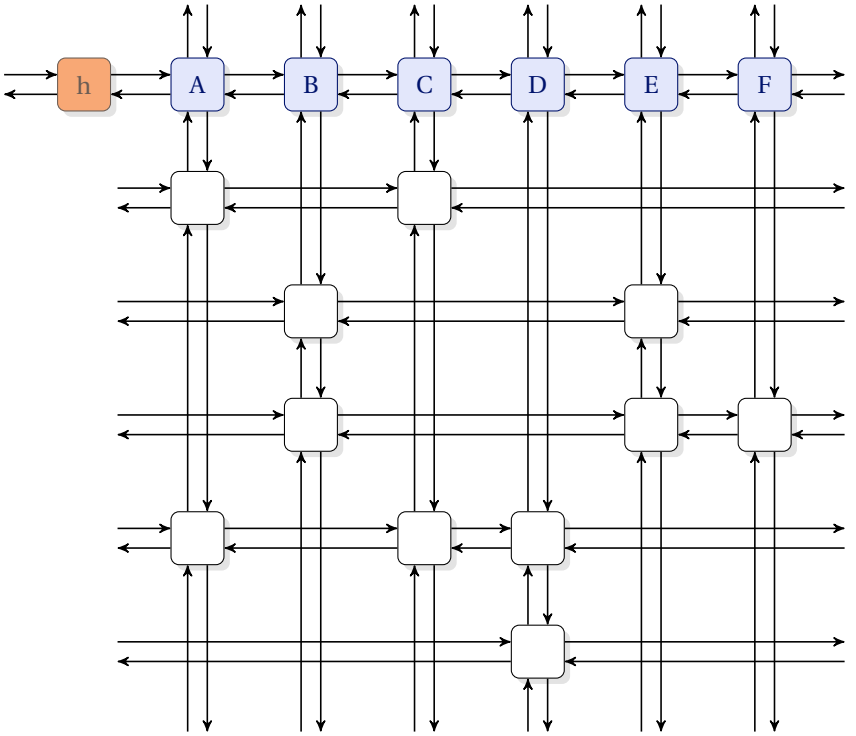
L'offre plaçant la reine sur cette case sera donc « `R2 C5 A7 BA` ».

Petite subtilité du problème cependant : si chaque ligne et chaque colonne doit être couverte une fois exactement, sept des quinze diagonales dans chaque sens n'ont *pas* à être couvertes. Ce n'est cependant pas un problème, il suffit d'ajouter des « offres » supplémentaires pour chaque diagnale, de sorte qu'on puisse toujours compléter avec ces offres les colonnes qui n'ont pas été occupées par le placement d'une reine. Le fichier « `reines.dlx` » décrit donc ce problème en terme de couverture exacte.

2 Liens dansants

2.1 Algorithme DLX

Après avoir décrit quelques problèmes, nous allons essayer de les résoudre! L'essentiel de l'algorithme, appelé DLX, proposé par D. Knuth, tient à une structure de données astucieusement pensée. Les 1 dans la matrice sont reliés, dans les deux directions, par des listes doublement chaînées circulaires afin de pouvoir naviguer aisément.



Outre les éléments correspondant aux 1 de la matrice, on ajoute dans chaque liste chaînée verticale un élément représentant les en-têtes de chaque colonne. Ces en-têtes de colonne sont inclus dans une liste chaînée horizontale, contenant un nœud supplémentaire, appelé `h`.

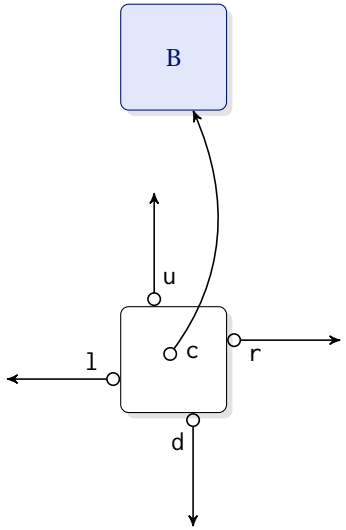
Lorsque l'on voudra fournir un problème donné à une fonction, on utilisera un pointeur vers cet élément `h`. À partir de cet élément, on peut en effet naviguer dans l'ensemble de la structure en se déplaçant dans les différentes listes chaînées.

Précisions que le schéma ci-dessus a une structure torique : les flèches qui arrivent tout en bas pointent en fait sur l'en-tête en haut de la même colonne et inversement, et il en est de même pour les flèches dont les extrémités sont à droite ou à gauche du schéma.

Pour mémoriser un nœud de cette structure, on utilisera la structure C suivante :

```c
struct node {
    struct node* u;
    struct node* d;
    struct node* l;
    struct node* r;
    struct node* c;
    char* name;
    int nb;
};
```

Les champs `u`, `d`, `l` et `r` contiendront les pointeurs vers les nœuds respectivement immédiatement au-dessus, en-dessous, à gauche et à droite du nœud décrit. Le champ `c` contiendra un pointeur vers l'en-tête de la colonne (pour les nœuds de l'en-tête, `h` compris, il s'agira d'un pointeur `NULL`).



Le champ `name` contiendra `NULL`, excepté pour les en-têtes de colonne, pour lesquels il désignera une chaîne nommant l'objet associé à la colonne en question. L'entier `nb` n'est pas utilisé pour le moment, excepté pour le nœud *h* qui contiendra le nombre d'objets à fournir (6 dans notre exemple).

On dispose d'une fonction `new_node` qui alloue et retourne un tel nœud (en cas d'échec d'allocation, la fonction affiche un message et arrête immédiatement le programme). Les champs pointeurs `u`, `d`, `l`, `r` et `c` sont initialisés avec l'adresse du nœud lui-même, le champ pointeur `name` avec l'adresse[2] d'une chaîne contenant `"Erreur"`, l'entier `nb` à zéro.

2. En principe, ils devraient être initialisés à `NULL`, mais nous allons éviter autant que possible les pointeurs `NULL` pour éviter les erreurs de segmentation.

2.2 Affichage d'une offre

On souhaite, pour un usage ultérieur, afficher les objets fournis par une offre donnée. On suppose que l'on dispose d'un pointeur vers un nœud `n` *quelconque* appartenant à l'offre en question, donc un nœud quelconque d'une des lignes.

1. Implémenter la fonction `print_line(Node* n)` prenant en argument un pointeur vers un nœud `n` et affiche les chaînes correspondant aux objets dans l'offre `n`, séparées par des espaces (l'affichage étant terminé par un retour à la ligne). On pourra utiliser « `printf("%s", str)` » pour afficher une chaîne de caractère `str`. Rappelons que les noms des objets sont mémorisés dans l'entête de la colonne, et que le champ `.c` de la structure `Node` permet d'accéder à l'entête de la colonne à laquelle appartient le nœud.

2. Tester la fonction `print_line` avec un appel à la fonction `test_print_line()` (celle-ci crée une structure similaire à celle décrite précédemment et appelle `print_line` sur un nœud d'une ligne. On doit obtenir l'affichage d'une phrase sans `"Erreur"`.

2.3 Lecture du fichier et construction de la structure

Nous allons à présent nous attacher à lire un fichier texte et construire la structure décrite précédemment en fonction des données qu'il contient. L'essentiel du travail de lecture est déjà fait dans la fonction `load_file`, qui attend un nom de fichier et retourne le nœud `h` de la structure construite à partir du fichier. En revanche, le travail de création des nœuds est incomplet, en particulier seuls les nœuds de la ligne d'en-tête sont créés, et les pointeurs restent à initialiser.

3. Compléter la lecture de la ligne d'en-tête (contenant la liste des différents objets) de manière à créer les nœuds correspondant aux en-têtes de colonne, en vous assurant que les pointeurs sont initalisés correctement. On ne touchera pas aux pointeurs `u`, `d` et `c`, et le pointeur `name` est déjà correctement initialisé avec une copie de la chaîne contenant le nom de chaque objet. Pour chaque nouveau nœud d'en-tête ajouté dans cette liste chaînée, il y a donc quatre pointeurs (`l` et |r|) à correctement mettre à jour : deux partant du nœud nouvellement ajouté, deux y pointant vers ce nœud.

4. Vérifier qu'un appel à « `print_line(h)` » affiche bien la liste de l'ensemble des objets, précédé de `"<>"` (qui est le nom associé par le programme au nœud `h`).

Pour la suite, remarquons que si `n` est un nœud de la ligne d'en-tête correspondant à un objet (donc autre que `h`), `n->d` désignera toujours le premier élément dans la colonne, et `n->u` le dernier (excepté tant que la liste est vide, auquel cas les deux pointeurs désigneront le nœud `n` lui-même).

Pour chaque ligne suivante dans le fichier, correspondant à une offre, un appel à `get_offer` remplit dans un premier temps un tableau `provides` de pointeurs vers les nœuds de la ligne d'en-tête correspondants aux objets de l'offre considérée. L'entier `nb_pr`

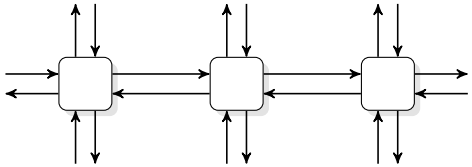
quant à lui contient le nombre d'objets (donc de pointeurs vers les nœuds correspondants) de l'offre.

5. Compléter le traitement d'une offre afin de créer les nœuds correspondant à cette offre, en prenant soin de bien initialiser les différents pointeurs concernés. Plutôt que d'allouer les `nb_pr` nœuds un à un, on peut allouer directement un tableau de `nb_pr` nœuds [3].

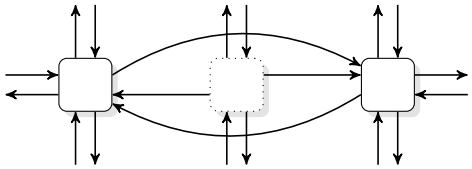
6. Vérifier qu'un appel à « `print_line(h->r->d)` » imprime correctement les objets `A` et `C`, et que `print_line(h->l->u->r)` affiche les objets `B`, `E` et `F` (dans un ordre quelconque).

2.4 Manipulation des listes circulaires

Les listes doublement chaînées présentent des particularités très intéressantes pour notre problème. On s'intéresse pour l'instant uniquement aux listes chaînées horizontales (pour un nœud `n`, ses voisins dans la liste sont donc `n->l` à gauche et `n->r` à droite).



7. Comprendre comment, pour une liste contenant au moins deux éléments, on peut retirer le nœud `n` de la liste chaînée circulaire en modifiant uniquement les valeurs de `n->l->r` et `n->r->l` comme illustré ci-dessous où le nœud central est « retiré » de la liste :



8. Modifier la fonction `remove_h(Node* n)` retirant le nœud `n` retirant le nœud `n` de la liste chaînée horizontale qui le contient. **On ne supprime pas le nœud en question, on se contente de l'extraire de la liste**, et on ne touche pas aux pointeurs du nœud `n` lui-même, c'est important pour la suite.

9. Vérifier le bon fonctionnement de la fonction `remove_h` avec le test fourni (on doit obtenir `F B` ou `B F`).

De façon plus remarquable, il est possible d'*annuler* une suppression !

10. Déterminer comment, connaissant un nœud `n` qui a été retiré d'une liste, on peut le replacer au même endroit qu'auparavant. On suppose que les pointeurs dans le reste de la liste sont exactement comme ils l'étaient juste après la suppression !

11. En déduire une implémentation de la fonction `replace_h(Node* n)` qui replace le nœud `n` dans la liste chaînée horizontale qu'il vient de quitter.

12. Vérifier le bon fonctionnement de cette fonction avec le test fourni (on doit retrouver `F B E` ou équivalent).

On remarquera que si l'on effectue une série de suppression, on peut toutes les annuler, à condition de le faire dans l'ordre inverse des suppressions.

13. Implémenter de même les fonctions `remove_v(Node* n)` et `replace_v(Node* n)` qui font de même pour les listes chaînées verticales.

2.5 Couverture d'une colonne

Supposons que l'on ait trouvé une offre qui fournissait un certain objet donné. Pour signifier qu'il n'est plus nécessaire de chercher cet objet, on va (éventuellement temporairement) « retirer » la colonne à laquelle il correspond de la liste chaînée (horizontale) des en-têtes. Mais ce n'est pas suffisant : toutes les offres qui fournissent ce même objet ne peuvent plus être choisies.

On va donc, pour chacune des offres fournissant l'objet en question (autrement dit, pour tous les nœuds dans la colonne concernée), on va retirer les nœuds leur correspondant des listes chaînées verticales des *autres* colonnes. Cela revient, en pratique, à supprimer de la matrice non seulement la colonne concernée mais également les lignes qui contiennent un 1 dans cette matrice. Par exemple, si l'on choisit une offre fournissant le premier objet, la matrice va être transformée de la façon suivante (la première colonne « disparaît », de même que les première et quatrième lignes qui fournissaient cet objet) :

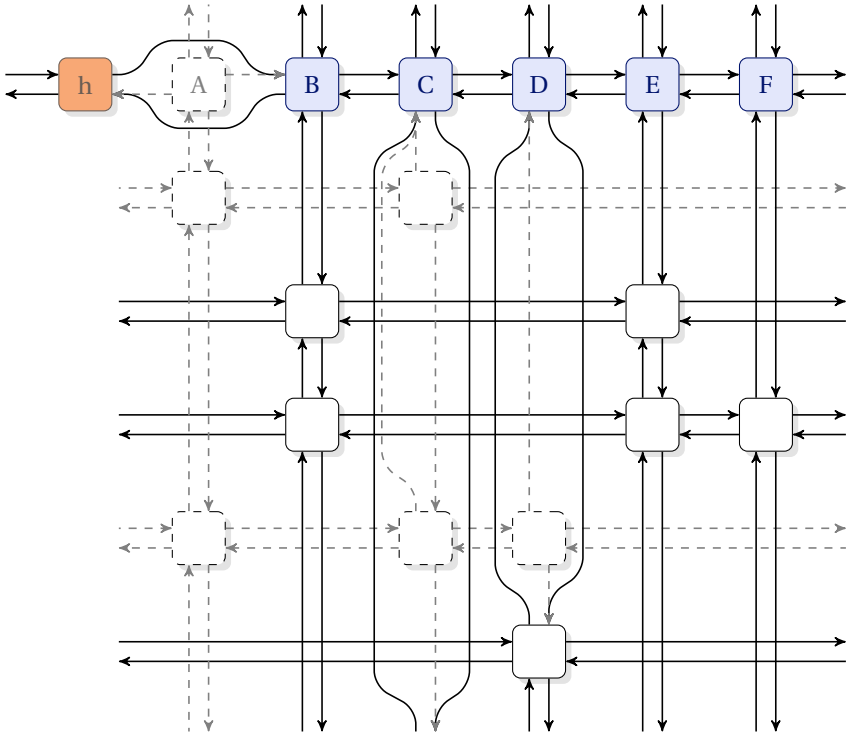
$$M = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \mapsto \quad \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Autrement dit, pour « couvrir » une colonne associée à un objet c (identifiée par un pointeur vers le nœud `n` correspondant à son en-tête), nous allons donc :

- « retirer » le nœud `n` d'en-tête correspondant à c de la liste doublement chaînée horizontale d'en-tête ;
- puis, pour tous les nœuds `r` dans la colonne sous `n` (distincts de `n` donc), considérés de haut en bas (que l'on obtient en visitant la liste doublement chaînée verticale partant de `n`) :
  - on considère tous les nœuds s distincts de r, en se déplaçant vers la droite en partant de r, dans la liste doublement chainée horizontale contenant r, et on les retire de leur liste doublement chaînée verticale (ce qui a pour effet pratique de « retirer » les lignes des offres fournissant l'objet c).

3. La seule différence pratique est qu'il faudra les libérer d'un seul coup avec un appel à `free`, mais on ne se préoccupe pas ici, exceptionnellement, de libérer la mémoire allouée, puisque le programme s'arrêtera après avoir listé les solutions qu'il a identifiées.

Voici ce que devient donc la structure après la couverture de la colonne A :



14. Proposer une fonction `cover(Node* c)` prenant en argument un pointeur vers un nœud en-tête d'une colonne et couvrant la colonne comme indiqué ci-dessus.

15. En utilisant le test fourni, tester la fonction `cover`, qui couvre deux colonnes successivement, celle de l'objet `C` puis celle de l'objet `E`, et effectue quelques affichages pour contrôler le résultat.

L'opération est inversible, à condition d'effectuer toutes les opérations dans l'ordre *inverse* de ce qui a été décrit précédemment. C'est ici que le caractère doublement chaîné va nous aider : on peut aisément parcourir les éléments vers le haut et vers la gauche.

16. En déduire une fonction `uncover(Node* c)` qui annule les effets d'un appel à la fonction `cover`.

17. En utilisant le test fourni, tester la fonction `uncover`, qui découvre la colonne de l'objet « `C` ». On doit retrouver les six objets et toutes les offres.

2.6 Utilisation pour la résolution avec retour sur trace

Nous avons à présent tous les outils qu'il nous faut pour énumérer les solutions à un problème. Pour ce faire, nous allons écrire une fonction `solve(h)` qui prend en argument un problème et affiche toutes ses solutions. Pour ce faire, elle alloue un tableau `output` de taille suffisante pour, éventuellement, contenir autant d'offres qu'il y a d'objets, puis appelle une fonction récursive `search(k, h, output)` qui va explorer toutes les possibilités de la façon suivante :

- si la liste des colonnes restant à couvrir est vide (tous les objets ont été fournis par les offres sélectionnées), appelle la fonction `print_solution(output, k)` pour afficher la solution courante, et retourne ;
- sinon, choisit une colonne *c* correspondant à un objet qui n'a pas encore été fourni, donc qui n'a pas encore été masquée, grace à la fonction `choose_column(h)` [4] ;
- couvre la colonne *c*
- puis, pour tout nœud `r` dans cette colonne :
  - sélectionne cette offre en plaçant un pointeur vers `r` dans `output[k]` ;
  - parcourt tous les autres nœuds `n` (distincts de `r`) dans la même liste doublement chaînée horizontale et couvre les colonnes auxquelles ils appartiennent (car les objets correspondants ont été fournis)
  - s'appelle récursivement avec `search(k+1, h, output)` ;
  - découvre les colonnes couvertes juste avant l'appel récursif, *en parcourant les nœuds de la liste horizontale dans l'ordre inverse* ;
- découvre la colonne *c*

On retrouvera l'article original de Donald Knuth, avec la démarche précédente en pseudo-code, à l'adresse `cdn.sci-phy.org/mp2i/Knuth-Dancing_links.pdf` .

2.7 Résolutions des problèmes

18. Compléter la fonction `search`.

19. Effectuer un premier test avec le fichier « `exemple.dlx` » qui correspond à l'exemple décrit au début du sujet. On doit retrouver les deux solutions attendues.

20. Utiliser le programme sur le problème du placement à table (`diner.dlx`), et déterminer le nombre de solutions à ce problème.

21. Faire de même avec le problème des 8 reines (`reines.dlx`).

4. Pour l'instant, l'implémentation fournie utilise la première colonne disponible, identifiée par `h->r`. Nous verrons plus tard comment améliorer ce choix.

3 Amélioration des performances

3.1 Limitations de l'approche précédente

Même si sur les exemples proposés l'algorithme effectue un travail quasi-instantané, les calculs peuvent être bien plus long dans des problèmes de plus grande taille. Si, plutôt que de vouloir lister *toutes* les solutions on n'en veut qu'une seule, nous avons vu qu'effectuer un choix pertinent présente des avantages.

Pour l'instant, nous avons choisi, à chaque étape, de fournir le premier objet qui n'ait pas encore été fourni, en sélectionnant directement la première colonne encore présente. Ce n'est pas toujours le choix le plus pertinent.

3.2 Améliorer le choix de colonne

Une possible meilleure solution est de choisir l'objet (la colonne) pour laquelle il y a le moins d'offres. Pour éviter que l'on ait à calculer la longueur de chaque liste chaînée verticale en permanence, on va utiliser le champ `nb` dans les nœuds en-tête de colonne pour mémoriser combien d'offres proposent l'objet associé à la colonne.

22. Modifier la fonction construisant la structure pour qu'elle renseigne le champ `nb` des en-têtes de colonne (on pourra soit le calculer à la fin de la construction, soit le tenir à jour à chaque ajout d'un nœud dans une colonne, au choix).

23. Modifier les fonctions `remove_v` et `replace_v` pour qu'elles tiennent à jour le champ `nb` dans l'en-tête de colonne (on rappelle que le champ `c` permet de retrouver l'en-tête d'une colonne depuis n'importe quel nœud de cette colonne).

24. Modifier la fonction `choose_column(Node* h)` pour qu'elle retourne non plus la première colonne mais celle correspondant à la plus petite liste doublement chaînée verticale (en cas d'égalité, on choisira la colonne librement parmi celles de longueur minimale).

25. Vérifier que le programme fonctionne toujours sur les exemples fournis.

4 Autres problèmes

4.1 Sudoku

26. Écrire un programme générant un fichier « `dlx` » décrivant le sudoku ci-dessous, et vérifier qu'il a une solution et une seule. Vous êtes libres d'écrire le programme générant le contenu du fichier `dlx` dans le langage de vitre choix.

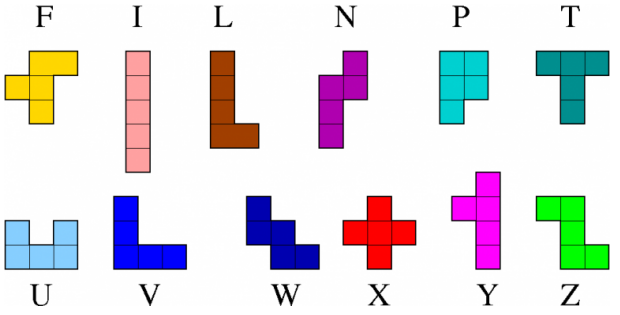
			8		1			
							4	3
5								
				7		8		
						1		
	2			3				
6							7	5
		3	4					
			2			6		

On s'inspirera du problème des 8 reines : il faut en effet placer un 1 dans chaque ligne et chaque colonne, mais on ne doit pas placer plusieurs 1 dans la même zone. Le programme générant le fichier `dlx` devra par ailleurs éliminer les offres qui ne sont pas compatibles avec les éléments déjà placés dans la grille.

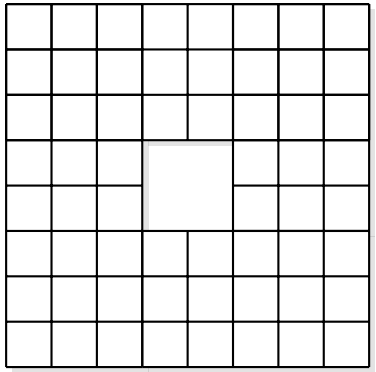
En théorie, en partant d'une grille vide, le programme peut énumérer toutes les grilles de sudoku possibles, mais il y en a une très grande quantité (six mille milliards de milliards environ), donc le programme mettra un peu de temps !

4.2 Puzzle

On dénombre 12 pentaminos, représentés ci-dessous :



On cherche à les placer de façon à remplir un carré de taille 8×8 dont les quatre cases centrales sont interdites. Attention, il est possible non seulement de tourner les pentaminos, mais également de les retourner.



27. Écrire un programme générant un fichier « `dlx` » décrivant le problème, et déterminer le nombre de solutions. On pourra réfléchir aux symétries du problème pour limiter la taille de l'espace à explorer.

On peut également essayer de trouver les solutions pour une grille de taille 3×20, 4×15, 5×12 ou 6×10.