Résolution du problème Exact Cover

Charles Bouillaguet

v1.0 du 12 mars 2021

Le but du projet est de paralléliser un programme séquentiel que nous vous fournissons et que ce document décrit. Le programme résoud une instance du problème de la couverture exacte, un problème d'optimisation combinatoire. Il s'agit d'un des 21 problèmes NP-complets de Karp [Kar72].

Grosso-modo, il y a un ensemble d'*objets* et il y a des *options* constituées d'un ou plusieurs objets. Le but du jeu consiste à choisir un sous-ensemble des *options* tel que chaque objet soit contenu dans une et une seule des options sélectionnées (on dit que chaque objet est *couvert* par une et une seule des options choisies).

Autrement dit, étant donné un ensemble U et une collection S de sous-ensembles de U, une *couverture exacte* de U est une sous-collection S^* de S telle que tout élément de U est élément d'exactement un des ensembles de S^*.

Il est commode de représenter S par une matrice M de 0 et de 1, dont les colonnes représentent les objets et où chaque ligne représente une option. L'option i contient l'objet j si $M_{ij} = 1$. Une solution est donc un vecteur $\mathbf{x}$ à coefficients 0/1 tel que $xM = (1, 1, \ldots, 1)$.

Le problème de la couverture exacte est surprenament versatile. Remplir une grille de Sudoku, par exemple, peut être vu comme une instance du problème de la couverture exacte. Plusieurs autres exemples sont donnés ci-dessous.

En fait, on considère ici un raffinement du problème. Certains objets sont *primaires* tandis que d'autres sont *secondaires*. Chaque option contient au moins un objet primaire. Les objets primaires doivent être couverts une fois *exactement*. Les objets secondaires doivent être couverts *au plus* une fois.

La référence absolue sur le sujet est [Knu19] (partiellement disponible en ligne). Le programme implante une légère variante de l'algorithme *Dancing Links* conçu par Knuth dans les années 1990. Il est capable de trouver *toutes* les solutions du problème, et donc de les afficher ou simplement de les compter.

Instances

Des instances du problème, de difficulté variable, sont mises à votre disposition et décrites ici. Certaines sont faites pour tester votre code sur de petits exemples. D'autres sont de véritables challenges, difficiles à résoudre.

Nombres de Bell

On pose $U = \{1, 2, \ldots, n\}$ et on considère que S est l'ensemble des parties non-vides de U. Une solution du problème est alors simplement une *partition* de U. Le nombre de solution est alors le nombre de manières dont U peut être partitionné, et il s'agit des *nombres de Bell* B_n. L'avantage c'est qu'on peut facilement les calculer, donc ça permet de vérifier que le code trouve le bon nombre de solutions.

n	B_n	T (s)
12	4 213 597	1.2
13	27 644 437	8
14	190 899 322	60

Couplages parfaits du graphe complet

Dans un graphe non-orienté, un *couplage* est un sous-ensemble des arêtes qui touche chaque sommet au plus une fois. Un couplage est parfait s'il touche tous les sommets. Trouver un couplage parfait est donc une instance du problème de la couverture exacte : on pose $U =$ l'ensemble des sommets, et S est formé de l'ensemble des paires de sommets. Les solutions de ce problème de couverture exacte sont précisément les couplages parfaits du graphe.

Dans le graphe complet à $2n$ sommets, il existe toujours des couplages parfaits ; leur nombre est connu et facile à calculer, et le programme peut donc les énumérer. La encore, ce n'est pas très intéressant mais ça permet de vérifier si le code fonctionne correctement.

n	#Solutions	T (s)
8	2 027 025	0.3
9	34 459 425	4.6
10	654 729 075	87

Carrés latins

Un *carré latin* est une sorte de Sudoku : dans une grille $n \times n$, il faut remplir chaque case avec les lettres A, B, C, …(il faut n lettres) de telle sorte que chaque ligne et chaque colonne contienne chaque lettre (une seule fois).

Former un carré latin peut être vu comme un problème de couverture exacte avec trois types d'objets :

- $(\alpha, i, \star)$ où α est une lettre et $1 \leq i \leq n$ un indice de ligne (ces objets forcent chaque ligne à contenir chaque lettre une et une seule fois).
- $(\alpha, \star, j)$ où α est une lettre et $1 \leq j \leq n$ un indice de colonne (ces objets forcent chaque colonne à contenir chaque lettre une et une seule fois).
- Enfin (i, j), avec $1 \leq i, j \leq n$ désigne une case de la grille. (ces objets forcent le remplissage de chaque case une et une seule fois).

L'option « placer la lettre α dans la case (i, j) » couvre les trois objets (i, j), $(\alpha, i, \star)$ et $(\alpha, \star, j)$. Il y a donc n^3 telles options pour $2n^2$ objets.

Un carré latin est *réduit* si les lettres apparaissent dans l'ordre sur la première ligne et la première colonne. On peut obtenir des carrés latins réduits en supprimant les options qui permettent d'autres

placements sur la première ligne/colonne. Il n'y a pas de formule qui donne le nombre de carrés latins réduits de taille n, mais ces nombres ont été calculés pour d'assez grandes valeurs de n [1].

n	#Solutions	T (s)
6	9 408	0.9
7	16 942 080	17
8	535 281 401 856	plusieurs heures

Carrés gréco-latins

Deux carrés latins de la même taille sont *orthogonaux* s'ils ne placent pas la même lettre dans la même case. Par exemple, si on écrit le deuxième avec des lettres grecques et qu'on superpose les deux :

Aα	Bβ	Cγ
Bγ	Cα	Aβ
Cβ	Aγ	Bα

On les appelle donc des carrés gréco-latins. Euler, le plus grand mathématicien du XVIII$^{\text{ème}}$ siècle, a démontré qu'on pouvait construire des carrés greco-latins de taille impaire, et de taille multiple de 4. Il conjectura également qu'on ne pouvait pas en construire de taille n si $n \equiv 2 \bmod 4$. Cette conjecture a été invalidée en 1959, quand une recherche par ordinateur en a exhibé un de taille 22. Un an plus tard, on en a trouvé un de taille 10.

[**challenge**] Ferez-vous aussi bien que les informaticiens des années 1960, sur le matériel de l'époque ? Trouvez un carré gréco-latin de taille 10 (environ 12h de calcul séquentiel).

Le livre *La Vie mode d'emploi* de Georges Perec (1978, prix Médicis la même année) est construit à partir d'un carré gréco-latin de taille 10.

Polyominos

Les polyominos sont des formes géométriques qu'on peut former en assemblant des petits carrés. Les *tetraminos* (de taille 4) sont en fait les pièces de Tetris. Les pentaminos (de taille 5) sont visibles figure 1. Il faut imaginer que ce sont des bouts de carton, et qu'on peut leur faire subir une *rotation* de 90 degrés (comme dans Tetris), ainsi qu'une *reflexion* (les « retourner »).

Voici une liste de questions « amusantes », par ordre de difficulté croissante. Combien de manière y a-t-il de faire tenir :

- Les 12 pentaminos dans un rectangle 10×6 ?
- Les 12 pentaminos *et* les 5 tetraminos dans deux rectangles de taille 4×10 ? (environ 10 minutes de calcul séquentiel)
- [**vrai challenge**] idem dans un rectangle de taille 8×10 (environ un mois de calcul séquentiel) ? Une des solutions est montrée figure 2.
- [**challenge**] Les 12 pentaminos et *un* tetramino dans un carré 8×8 (environ 12h de calcul séquentiel) ?

1. `https://oeis.org/A000315`

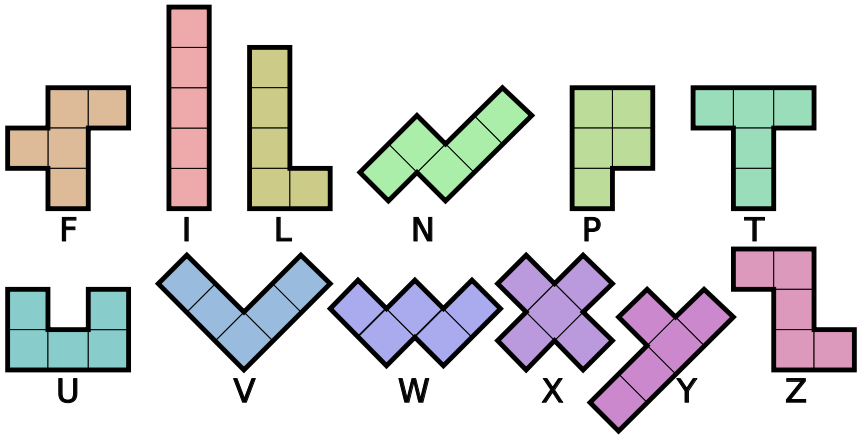

FIGURE 1 – Les 12 pentaminos.

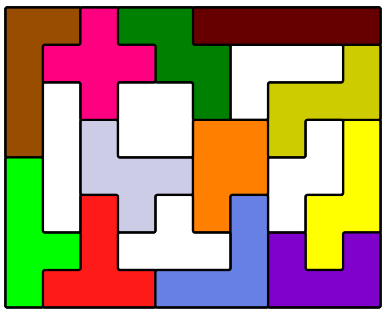

FIGURE 2 – Les 12 pentaminos et les 5 tetraminos dans un rectangle 8×10.

Travail à effectuer

Il y a plusieurs « sous-tâches » que vous pouvez accomplir plus ou moins dans le désordre.

1. Parallélisation avec OpenMP uniquement sur une machine multi-coeurs.
2. Parallélisation avec MPI uniquement sur un *cluster*.
3. Parallélisation avec MPI + OpenMP : on lance un seul processus MPI par noeud et on fait du multi-thread à l'intérieur (c'est le mieux a priori).
4. *Checkpointing* : il faut que si le calcul parallèle s'arrête (panne réseau, plantage, coupure electrique, ...) , on ne soit pas obligé de tout recommencer depuis le début. Pour cela, une solution consiste à sauvegarder périodiquement des *checkpoints*, et de pouvoir repartir du dernier checkpoint. Votre implantation parallèle peut par exemple sauvegarder un *checkpoint* chaque minute.
5. Résolution des challenges, le plus vite possible.

Vous noterez par ailleurs les points suivants :

1. *Quoi que vous fassiez*, vous **devez** :
   (a) *Mesurer* les performance obtenues (notamment l'accélération atteinte par rapport au code séquentiel, lorsque ça a un sens) sur *plusieurs* instances du problème.
   (b) *Commenter* ces résultats : sont-ils bons ou pas ? S'ils sont mauvais, pourquoi ? En est-on réduit à émettre des hypothèses ou bien peut-on les confirmer par une expérience ?
2. Il n'est pas malin de charger la matrice depuis le système de fichiers sur *tous* les processus à la fois : avec beaucoup de processus, ceci pourrait saturer le serveur de fichier.
3. Pour le checkpointing : il suffit de conserver le *dernier* checkpoint. Il faut cependant faire attention au fait que ça peut planter *pendant* son écriture (indice : dans les OS POSIX, l'appel système `rename` peut remplacer un fichier de manière atomique).

Il n'est pas obligatoire de tout faire pour avoir la moyenne, surtout vu les circonstances exceptionnelles causées par l'épidémie de COVID-19. Pour viser la note maximale, vous mettrez en oeuvre le plus d'améliorations possibles.

Travail à remettre

La date limite de rendu est fixée au dimanche 30 mai 2021 à 23 :59.

Vous *pouvez*, mais ce n'est pas obligatoire, nous faire un rendu intermédiaire de votre code avec un bref rapport (suffisamment pour qu'on comprenne ce que vous avez fait et quel problèmes vous avez, suffisamment peu pour que ça ne vous prenne pas longtemps). Ça peut être n'importe quand.

Que ce soit pour le rendu intermédiaire (facultatif) ou le rendu final (obligatoire), vous devrez remettre le code source, sous la forme d'une archive `tar.gz` compressée. L'archive ne doit contenir aucun exécutable, et les différentes versions demandées devront être localisées dans des répertoires différents. Chaque répertoire devra contenir un fichier `Makefile` : la commande `make` devra permettre de lancer la compilation.

Votre propre code doit compiler sans avertissements, même avec les options `-Wall -Wextra`.

Un `Makefile` situé à la racine de votre projet devra permettre (avec la commande `make`) de lancer la compilation de chaque version.

Pour le rendu final, vous devrez écrire un rapport au format `PDF` (de 5 à 10 pages présentant vos algorithmes, vos choix d'implantation (sans code source), vos résultats (notamment vos efficacités parallèles) et vos conclusions. L'analyse du comportement de vos programmes sera particulièrement appréciée. Le rapport doit indiquer clairement quelles machines ont été utilisées pour les tests de performance.

Quelques précisions importantes

- Le projet est à réaliser par petits groupes (taille ≤ 4).
- Vous **devez** lire les conditions d'utilisation de Grid'5000. Vous veillerez notamment à ne pas violer la charte. Des manquements répétés seront sanctionnés.
- Le rendu se fera sur Moodle.
- En cas d'imprévu ou de problème technique commun, n'hésitez pas à nous contacter pour que nous puissions vous proposer une solution ou une alternative.

Et maintenant, on passe aux choses sérieuses.

Propagande

La *programmation littéraire* (ou programmation lettrée) consiste à ecrire un programme dans le but qu'il soit lu par d'autres êtres humains. Pour cela, le code est découpé en petits morceaux, et entrelacé avec des explications [Knu92]. Cette technique a été utilisée pour écrire le solveur séquentiel qui est mis à votre disposition (et il était quasi-correct du premier coup). Plus précisement, le programme `noweb` [Ram94] permet d'extraire à la fois le code C et ce document LaTeX depuis le même fichier `.nw`.

1 Structure générale du programme séquentiel

Le programme est constitué d'un unique fichier C, qui a la stucture suivante :

7a ⟨* 7a⟩≡
⟨*Inclusion des en-têtes* 7b⟩
⟨*Variables globales* 8b⟩
⟨*Fonctions auxiliaires* 8c⟩
⟨*Coeur de l'algorithme* 17a⟩
⟨*Fonction* `main` 8a⟩

Commençons par le commencement, il nous faut les en-têtes habituels.

7b ⟨*Inclusion des en-têtes* 7b⟩≡ (7a)
```
#include <ctype.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <err.h>
#include <getopt.h>
#include <sys/time.h>
```

Et sans plus attendre, voici la fonction `main`. Elle charge une *instance* du problème dans une variable `instance`, puis elle initialise un *contexte* qui contient l'état actuel de la procédure de recherche. Enfin, elle lance la recherche. La fonction `load_matrix` est décrite section 6, tandis que `backtracking_setup` et `solve` sont décrites section 5.

8a ⟨*Fonction* `main` 8a⟩≡ (7a)

```
int main(int argc, char **argv)
{
        ⟨Traitement des options en ligne de commande 10⟩
        struct instance_t * instance = load_matrix(in_filename);
        struct context_t * ctx = backtracking_setup(instance);
        start = wtime();
        solve(instance, ctx);
        printf("FINI. Trouvé %lld solutions en %.1fs\n", ctx->solutions,
                        wtime() - start);
        exit(EXIT_SUCCESS);
}
```

La variable `start` est une variable globale, car d'autres fonctions qui font du *reporting* ont besoin de savoir combien de temps s'est écoulé depuis le début du calcul.

8b ⟨*Variables globales* 8b⟩≡ (7a) 9b▷

```
double start = 0.0;
```

Pour mesurer le passage du temps, on utilise `gettimeofday`, dans une version équivalente à `MPI_Wtime` ou `omp_get_wtime`.

8c ⟨*Fonctions auxiliaires* 8c⟩≡ (7a) 9a▷

```
double wtime()
{
        struct timeval ts;
        gettimeofday(&ts, NULL);
        return (double) ts.tv_sec + ts.tv_usec / 1e6;
}
```

2 Options en ligne de commande

L'argument principal à récupérer est le nom du fichier qui contient la matrice (argument `--in`). De plus, on peut spécifier s'il faut seulement compter toutes les solutions, ou bien les afficher au fur-et-à-mesure qu'on les trouve. De plus, on peut paramétrer la verbosité du processus. En cas de problème, on affiche le mode d'emploi :

9a ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁8c 11b▷

```
void usage(char **argv)
{
        printf("%s --in FILENAME [OPTIONS]\n\n", argv[0]);
        printf("Options:\n");
        printf("--progress-report N   display a message every N nodes (0 to disable)\n");
        printf("--print-solutions     display solutions when they are found\n");
        printf("--stop-after N        stop the search once N solutions are found\n");
        exit(0);
}
```

9b ⟨*Variables globales* 8b⟩+≡ (7a) ◁8b 11a▷

```
char *in_filename = NULL;              // nom du fichier contenant la matrice
bool print_solutions = false;          // affiche chaque solution
long long report_delta = 1e6;          // affiche un rapport tous les ... noeuds
long long next_report;                 // prochain rapport affiché au noeud...
long long max_solutions = 0x7fffffffffffffff;        // stop après ... solutions
```

Le traitement des options utilise la fonction `getopt_long`, qui est une extension GNU de `getopt`.

10 ⟨ *Traitement des options en ligne de commande* 10⟩≡ (8a)

```
struct option longopts[5] = {
        {"in", required_argument, NULL, 'i'},
        {"progress-report", required_argument, NULL, 'v'},
        {"print-solutions", no_argument, NULL, 'p'},
        {"stop-after", required_argument, NULL, 's'},
        {NULL, 0, NULL, 0}
};
char ch;
while ((ch = getopt_long(argc, argv, "", longopts, NULL)) != -1) {
        switch (ch) {
        case 'i':
                in_filename = optarg;
                break;
        case 'p':
                print_solutions = true;
                break;
        case 's':
                max_solutions = atoll(optarg);
                break;
        case 'v':
                report_delta = atoll(optarg);
                break;
        default:
                errx(1, "Unknown option\n");
        }
}
if (in_filename == NULL)
        usage(argv);
next_report = report_delta;
```

3 Représentation en mémoire d'une instance du problème

Les objets sont représentés par des entiers $(0, 1, 2, \dots)$, et les options sont des listes d'entiers. Les objets primaires sont les premiers. Le tout est décrit par la structure de données suivante. La seule subtilité est le stockage des options (au format « *Compressed Sparse Row* »). Les listes d'entiers qui correspondent aux différentes options sont contigües en mémoire (dans le champ `options`), tandis que `ptr` indique où commence et où finit chaque option. Le champ `item_name` est facultatif (il n'est pas nécessaire à la résolution du problème).

Les options, elles aussi, sont numérotées à partir de zéro.

11a ⟨*Variables globales* 8b⟩+≡ (7a) ◁9b 12a▷

```
struct instance_t {
        int n_items;
        int n_primary;
        int n_options;
        char **item_name;   // potentiellement NULL, sinon de taille n_items
        int *options;       // l'option i contient les objets options[ptr[i]:ptr[i+1]]
        int *ptr;           // taille n_options + 1
};
```

Les `n_primary` premiers objets sont primaires. C'est comme ça qu'on les reconnait.

11b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁9a 11c▷

```
bool item_is_primary(const struct instance_t *instance, int item)
{
        return item < instance->n_primary;
}
```

Avant de rentrer dans le détail du code qui fournit ces données, il est utile de pouvoir les afficher. Ceci illustre le stockage des options.

11c ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁11b 12b▷

```
void print_option(const struct instance_t *instance, int option)
{
        if (instance->item_name == NULL)
                errx(1, "tentative d'affichage sans noms d'objet");
        for (int p = instance->ptr[option]; p < instance->ptr[option + 1]; p++) {
                int item = instance->options[p];
                printf("%s ", instance->item_name[item]);
        }
        printf("\n");
}
```

4 Tableaux creux

L'algorithme repose d'une manière essentielle sur une structure de donnée particulière, les *tableaux creux*. Il s'agit d'un moyen de représenter des sous-ensembles de $\{0, 1, \ldots, n-1\}$, avec les deux caractéristiques suivantes :

— Ajout et suppression d'un élément en temps constant (indépendant de n).
— Test d'appartenance en temps constant.
— Iteration efficace sur les éléments (temps constant par élément).

On peut obtenir ces résultats avec des listes doublement chainées (c'est comme ça que c'était fait dans l'algorithme original et c'est ce qui lui a donné le nom de *Dancing Links*). La technique des tableaux creux donne les mêmes résultats. Il s'agit d'une vieille idée, décrite dans l'exercice 2.12 du livre d'algorithmique classique *The Design and Analysis of Computer Algorithms* de Aho, Hopcroft, et Ullman (Addison–Wesley, 1974). Elle est couramment employée dans le contexte de l'algèbre linéaire creuse.

L'idée consiste à représenter un sous-ensemble S de $0, \ldots, n-1$ par deux tableaux p et q ainsi qu'un entier n. L'ensemble S est de taille k, et contient les entiers $p[0], \ldots, p[k-1]$. Si $x \in S$, alors x est l'un des $p[i]$ pour $0 \leq i < k$, et on doit avoir $q[x] = i$. On fixe que $q[x] \geq k$ lorsque x n'est pas dans l'ensemble S. Autrement dit, p indique *quels* éléments contient le sous-ensemble, tandis que q indique *où* ils sont (dans p).

12a ⟨*Variables globales* 8b⟩+≡ (7a) ◁11a 15a▷

```
struct sparse_array_t {
        int len;          // nombre d'éléments stockés
        int capacity;     // taille maximale
        int *p;           // contenu de l'ensemble = p[0:len]
        int *q;           // taille capacity (tout comme p)
};
```

12b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁11c 13a▷

```
struct sparse_array_t * sparse_array_init(int n)
{
        struct sparse_array_t *S = malloc(sizeof(*S));
        if (S == NULL)
                err(1, "impossible d'allouer un tableau creux");
        S->len = 0;
        S->capacity = n;
        S->p = malloc(n * sizeof(int));
        S->q = malloc(n * sizeof(int));
        if (S->p == NULL || S->q == NULL)
                err(1, "Impossible d'allouer p/q dans un tableau creux");
        for (int i = 0; i < n; i++)
                S->q[i] = n;          // initialement vide
        return S;
}
```

Le test d'appartenance est très facile, puisqu'il suffit de regarder dans q.

13a ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁12b 13b▷

```
bool sparse_array_membership(const struct sparse_array_t *S, int x)
{
        return (S->q[x] < S->len);
}
```

13b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁13a 13c▷

```
bool sparse_array_empty(const struct sparse_array_t *S)
{
        return (S->len == 0);
}
```

Pour ajouter un élément, il faut incrémentter `len`, et mettre à jour les cases correspondantes de p et q. Ceci suppose que x n'appartient pas déjà à l'ensemble.

13c ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁13b 13d▷

```
void sparse_array_add(struct sparse_array_t *S, int x)
{
        int i = S->len;
        S->p[i] = x;
        S->q[x] = i;
        S->len = i + 1;
}
```

Retirer un élément x d'un tableau creux n'est pas beaucoup plus compliqué, mais il faut permuter des éléments de p pour « pousser » x vers la fin et mettre à jour q en conséquence. Ceci suppose que x appartient bien à l'ensemble.

13d ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁13c 14▷

```
void sparse_array_remove(struct sparse_array_t *S, int x)
{
        int j = S->q[x];
        int n = S->len - 1;
        // échange p[j] et p[n]
        int y = S->p[n];
        S->p[n] = x;
        S->p[j] = y;
        // met q à jour
        S->q[x] = n;
        S->q[y] = j;
        S->len = n;
}
```

Un gros avantage de la procédure précédente, c'est que si on vient d'ajouter/retirer un élément d'un tableau creux, c'est très facile d'annuler l'effet de cette opération, même si on ne sait pas quel élément est concerné.

14 ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁13d 15b▷

```
void sparse_array_unremove(struct sparse_array_t *S)
{
        S->len++;
}

void sparse_array_unadd(struct sparse_array_t *S)
{
        S->len--;
}
```

5 Algorithme de recherche

À tout instant, on maintient un ensemble d'objets *actifs*. Un objet est actif s'il faut le couvrir. Pour chaque objet actif, on maintient une liste d'options *actives* qui le contiennent. Une options est active si elle contient uniquement des objets actifs. Par conséquent, à n'importe quel moment, on peut choisir d'utiliser une des options actives.

L'algorithme repose sur l'opération qui consiste à *couvrir* un objet. Pour couvrir un objet, on le retire de la liste des objets actifs, et on *désactive* toutes les options qui le contiennent.

L'algorithme est alors le suivant :

- Choisir un objet actif o. S'il n'y en a plus, alors arrêter : on a trouvé une solution.
- Si aucune option active ne contient o, alors arrêter : il est impossible de couvrir o et on est dans une impasse.
- Couvrir l'objet o.
- Pour chaque option $\mathcal{O}$ qui contenait o (c'étaient les options actives associées à o), *choisir* l'option $\mathcal{O}$: couvrir tous les objets $o' \neq o$ dans $\mathcal{O}$, puis résoudre récursivement le sous-problème ainsi obtenu.

Du coup, l'algorithme explore un arbre de recherche, où chaque noeud est étiqueté avec l'objet choisi, et où traverser une arête revient à choisir une des options qui couvre cet objet. L'arbre est exploré en profondeur d'abord. Il s'agit d'un exemple classique de *backtracking*.

On utilise des tableaux creux pour stocker toutes ces listes. On encapsule le tout dans un « objet » `context_t` nommé `ctx`.

15a ⟨ *Variables globales* 8b⟩+≡ (7a) ◁12a 20b▷

```
struct context_t {
        struct sparse_array_t *active_items;      // objets actifs
        struct sparse_array_t **active_options;   // options actives contenant l'objet i
        int *chosen_options;                      // options choisies à ce stade
        int *child_num;                           // numéro du fils exploré
        int *num_children;                        // nombre de fils à explorer
        int level;                                // nombre d'options choisies
        long long nodes;                          // nombre de noeuds explorés
        long long solutions;                      // nombre de solutions trouvées
};
```

Un objet est actif s'il appartient à la la liste des objets actifs. Donc, tester si un objet est actif revient à un test d'appartenance au tableau creux qui stocke les objets actifs.

15b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁14 16▷

```
bool item_is_active(const struct context_t *ctx, int item)
{
        return sparse_array_membership(ctx->active_items, item);
}
```

Il faut noter qu'à n'importe quel moment, le contexte est entièrement décrit par les options choisies jusqu'ici. Par conséquent, si on veut *sérialiser* simplement un contexte (pour le transmettre sur un réseau où le stocker dans un fichier), il suffit de donner la liste des options choisies.

Commençons par le plus facile : que faire quand on a trouvé une solution ?

16 ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁15b 19a▷

```
void solution_found(const struct instance_t *instance, struct context_t *ctx)
{
        ctx->solutions++;
        if (!print_solutions)
                return;
        printf("Trouvé une nouvelle solution au niveau %d après %lld noeuds\n",
                        ctx->level, ctx->nodes);
        printf("Options : \n");
        for (int i = 0; i < ctx->level; i++) {
                int option = ctx->chosen_options[i];
                printf("+ %d : ", option);
                print_option(instance, option);
        }
        printf("\n");
        printf("-----------------------------------------------------\n");
}
```

Le « contexte », c'est-à-dire l'état courant de la procédure de recherche, doit être initalisé avant toute autre opération. Il faut allouer les tableaux et initialiser les variables du `context_t`.

17a ⟨*Coeur de l'algorithme* 17a⟩≡ (7a) 18a▷

```
struct context_t * backtracking_setup(const struct instance_t *instance)
{
        struct context_t *ctx = malloc(sizeof(*ctx));
        if (ctx == NULL)
                err(1, "impossible d'allouer un contexte");
        ctx->level = 0;
        ctx->nodes = 0;
        ctx->solutions = 0;
        int n = instance->n_items;
        int m = instance->n_options;
        ctx->active_options = malloc(n * sizeof(*ctx->active_options));
        ctx->chosen_options = malloc(n * sizeof(*ctx->chosen_options));
        ctx->child_num = malloc(n * sizeof(*ctx->child_num));
        ctx->num_children = malloc(n * sizeof(*ctx->num_children));
        if (ctx->active_options == NULL || ctx->chosen_options == NULL
                || ctx->child_num == NULL || ctx->num_children == NULL)
                err(1, "impossible d'allouer le contexte");
        ⟨Initialise les objets actifs 17b⟩
        ⟨Initialise les options actives 17c⟩
        return ctx;
}
```

Initialement, tous les objets primaires sont actifs. Un objet secondaire ne l'est jamais.

17b ⟨*Initialise les objets actifs* 17b⟩≡ (17a)

```
ctx->active_items = sparse_array_init(n);
for (int item = 0; item < instance->n_primary; item++)
        sparse_array_add(ctx->active_items, item);
```

Au début, toutes les options sont actives.

17c ⟨*Initialise les options actives* 17c⟩≡ (17a)

```
for (int item = 0; item < n; item++)
        ctx->active_options[item] = sparse_array_init(m);
for (int option = 0; option < m; option++)
        for (int k = instance->ptr[option]; k < instance->ptr[option + 1]; k++) {
                int item = instance->options[k];
                sparse_array_add(ctx->active_options[item], option);
        }
```

Pour résoudre le problème, on choisit l'objet le plus dur à couvrir (celui qui est couvert par le moins d'options encore actives). On essaye de le couvrir avec chacune des options possibles (celles qui sont encore actives). Les fonctions `cover(...)` et `uncover(...)` sont décrites plus bas. Grossomodo, `uncover(instance, ctx, i)` restaure `ctx` dans l'état où il se trouvait juste avant que `cover(instance, ctx, i)` ne soit appelé.

18a ⟨*Coeur de l'algorithme* 17a⟩+≡ (7a) ◁17a

```
void solve(const struct instance_t *instance, struct context_t *ctx)
{
        ctx->nodes++;
        if (ctx->nodes == next_report)
                progress_report(ctx);
        if (sparse_array_empty(ctx->active_items)) {
                solution_found(instance, ctx);
                return;                         /* succès : plus d'objet actif */
        }
        int chosen_item = choose_next_item(ctx);
        struct sparse_array_t *active_options = ctx->active_options[chosen_item];
        if (sparse_array_empty(active_options))
                return;           /* échec : impossible de couvrir chosen_item */
        cover(instance, ctx, chosen_item);
        ⟨Branchement sur les options actives de l'objet choisi 18b⟩
        uncover(instance, ctx, chosen_item);                    /* backtrack */
}
```

18b ⟨*Branchement sur les options actives de l'objet choisi* 18b⟩≡ (18a)

```
ctx->num_children[ctx->level] = active_options->len;
for (int k = 0; k < active_options->len; k++) {
        int option = active_options->p[k];
        ctx->child_num[ctx->level] = k;
        choose_option(instance, ctx, option, chosen_item);
        solve(instance, ctx);
        if (ctx->solutions >= max_solutions)
                return;
        unchoose_option(instance, ctx, option, chosen_item);
}
```

Commençons par spécifier ce en quoi consiste le faire de « *choisir* » `option` dans le but de couvrir `item`. En fait, ça ne fait que *couvrir* tous les objets que `option` contient *sauf* `chosen_item`, car on l'a déjà couvert manuellement dans `solve()`.

19a ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁16 19b▷

```
void cover(const struct instance_t *instance, struct context_t *ctx, int item);

void choose_option(const struct instance_t *instance, struct context_t *ctx,
                        int option, int chosen_item)
{
        ctx->chosen_options[ctx->level] = option;
        ctx->level++;
        for (int p = instance->ptr[option]; p < instance->ptr[option + 1]; p++) {
                int item = instance->options[p];
                if (item == chosen_item)
                        continue;
                cover(instance, ctx, item);
        }
}
```

« *Déchoisir* » l'option, c'est restaurer le contexte `ctx` dans l'état où il se trouvait avant qu'on appelle `choose_option(instance, ctx, option, chosen_item)`. Pour cela, il suffit de faire les mêmes opérations en sens inverse, c'est-à-dire découvrir les objets couverts (dans l'ordre inverse).

19b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁19a 20a▷

```
void uncover(const struct instance_t *instance, struct context_t *ctx, int item);

void unchoose_option(const struct instance_t *instance, struct context_t *ctx,
                        int option, int chosen_item)
{
        for (int p = instance->ptr[option + 1] - 1; p >= instance->ptr[option]; p--) {
                int item = instance->options[p];
                if (item == chosen_item)
                        continue;
                uncover(instance, ctx, item);
        }
        ctx->level--;
}
```

À chaque étape, on choisir l'objet « le plus dur » à couvrir, c.a.d. celui qui a le moins d'options actives (ceci réduit le facteur de branchement). Ceci implante l'heuristique MRV (*Minimum Remaining Values*), qui est très efficace. En cas d'égalité, on fait un peu ce qu'on veut.

20a ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁19b 21a▷

```
int choose_next_item(struct context_t *ctx)
{
        int best_item = -1;
        int best_options = 0x7fffffff;
        struct sparse_array_t *active_items = ctx->active_items;
        for (int i = 0; i < active_items->len; i++) {
                int item = active_items->p[i];
                struct sparse_array_t *active_options = ctx->active_options[item];
                int k = active_options->len;
                if (k < best_options) {
                        best_item = item;
                        best_options = k;
                }
        }
        return best_item;
}
```

Afficher un rapport de progresion est important, surtout si le calcul prend vraiment longtemps. On essaye d'afficher où l'algorithme en est dans l'exploration de l'arbre de recherche, en affichant le chemin depuis la racine jusqu'au noeud courant. Pour chaque noeud traversé, on affiche le nombre d'enfants et le numéro de l'enfant actuellement choisi. On saute les noeuds qui n'ont qu'un seul fils. Pour afficher ça ne manière compacte, on utilise une représentation alphanumérique : le fils i sur n est affiché en deux caractères : `DIGITS[i]` suivi de `DIGITS[n]`.

20b ⟨*Variables globales* 8b⟩+≡ (7a) ◁15a

```
static const char DIGITS[62] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                                'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
                                'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
                                'u', 'v', 'w', 'x', 'y', 'z',
                                'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
                                'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
                                'U', 'V', 'W', 'X', 'Y', 'Z'};
```

21a ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁20a 21b▷

```
void progress_report(const struct context_t *ctx)
{
        double now = wtime();
        printf("Exploré %lld noeuds, trouvé %lld solutions, temps écoulé %.1fs. ",
                        ctx->nodes, ctx->solutions, now - start);
        int i = 0;
        for (int k = 0; k < ctx->level; k++) {
                if (i > 44)
                        break;
                int n = ctx->child_num[k];
                int m = ctx->num_children[k];
                if (m == 1)
                        continue;
                printf("%c%c ", (n < 62) ? DIGITS[n] : '*', (m < 62) ? DIGITS[m] : '*');
                i++;
        }
        printf("\n"),
        next_report += report_delta;
}
```

À ce stade, il reste à implanter les opérations de couverture/découverture d'un objet.

Pour couvrir un objet, on le retire de la liste des objets actifs (il en fait partie si et seulement s'il est primaire). Ensuite, on *désactive* toutes les options (actives) qui le contiennent, car elles ne sont plus utilisables.

21b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁21a 22a▷

```
void deactivate(const struct instance_t *instance, struct context_t *ctx,
                        int option, int covered_item);

void cover(const struct instance_t *instance, struct context_t *ctx, int item)
{
        if (item_is_primary(instance, item))
                sparse_array_remove(ctx->active_items, item);
        struct sparse_array_t *active_options = ctx->active_options[item];
        for (int i = 0; i < active_options->len; i++) {
                int option = active_options->p[i];
                deactivate(instance, ctx, option, item);
        }
}
```

Le fait de désactiver `option` pour couvrir `covered_item` retire l'option de toutes les listes d'options actives où elle apparaissait. Comme l'option était active, alors tous les objets qu'elle contient sont actif, *sauf* celui qu'on est en train de couvrir qui a déjà été désactivé dans `cover()`.

22a ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁21b 22b▷

```
void deactivate(const struct instance_t *instance, struct context_t *ctx,
                        int option, int covered_item)
{
        for (int k = instance->ptr[option]; k < instance->ptr[option+1]; k++) {
                int item = instance->options[k];
                if (item == covered_item)
                        continue;
                sparse_array_remove(ctx->active_options[item], option);
        }
}
```

Il nous faut maintenant être capable de faire les opérations inverses. L'idée générale est la même qu'avant :

- `reactivate(instance, ctx, option, uncovered_item)` remet le contexte dans l'état où il était juste avant `reactivate(instance, ctx, option, uncovered_item)`.
- `uncover(instance, ctx, item)` remet le contexte dans l'état où il était juste avant `cover(instance, ctx, item)`.

Il faut faire attention à défaire tout ce qu'on a fait dans l'ordre inverse.

22b ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁22a 23▷

```
void reactivate(const struct instance_t *instance, struct context_t *ctx,
                        int option, int uncovered_item);

void uncover(const struct instance_t *instance, struct context_t *ctx, int item)
{
        struct sparse_array_t *active_options = ctx->active_options[item];
        for (int i = active_options->len - 1; i >= 0; i--) {
                int option = active_options->p[i];
                reactivate(instance, ctx, option, item);
        }
        if (item_is_primary(instance, item))
                sparse_array_unremove(ctx->active_items);
}
```

23 ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁22b 24▷

```
void reactivate(const struct instance_t *instance, struct context_t *ctx,
                        int option, int uncovered_item)
{
        for (int k = instance->ptr[option + 1] - 1; k >= instance->ptr[option]; k--) {
                int item = instance->options[k];
                if (item == uncovered_item)
                        continue;
                sparse_array_unremove(ctx->active_options[item]);
        }
}
```

6 Lecture de l'instance du problème

Il s'agit de lire la matrice 0/1 décrite dans l'introduction, qui est a priori très creuse, depuis un fichier texte. On a gardé ça pour la fin, car c'est un peu compliqué et ce n'est pas le plus intéressant.

— Chaque objet possède un identifiant, formé d'une chaine de caractères non-blancs (lettres, chiffres et underscore autorisés).
— La première ligne du fichier contient le nombre d'objets et le nombre d'options.
— La deuxième ligne du fichier contient les noms de tous les objets primaires séparés par des espaces, suivis du caractère `|`, suivi des noms des objets secondaires, séparés par des espaces. Le `|` est facultatif s'il n'y a pas d'objets secondaires.
— Chacune des lignes suivantes décrit une option : c'est la liste des noms des objets qu'elle contient, séparés par des espaces.

Voici par exemple un fichier valide.

```
7 6
A B C D E | F G
C E F
A D G
B C F
A D
B G
D E G
```

24 ⟨*Fonctions auxiliaires* 8c⟩+≡ (7a) ◁23

```
struct instance_t * load_matrix(const char *filename)
{
        struct instance_t *instance = malloc(sizeof(*instance));
        if (instance == NULL)
                err(1, "Impossible d'allouer l'instance");
        FILE *in = fopen(filename, "r");
        if (in == NULL)
                err(1, "Impossible d'ouvrir %s en lecture", filename);
        ⟨Lit les tailles du problème et alloue la mémoire 25a⟩
        ⟨Gestion des I/O 25b⟩
        ⟨Lit les noms des objets 26b⟩
        ⟨Lit les options 28⟩
        fclose(in);
        fprintf(stderr, "Lu %d objets (%d principaux) et %d options\n",
                instance->n_items, instance->n_primary, instance->n_options);
        return instance;
}
```

25a ⟨*Lit les tailles du problème et alloue la mémoire* 25a⟩≡ (24)

```
int n_it, n_op;
if (fscanf(in, "%d %d\n", &n_it, &n_op) != 2)
        errx(1, "Erreur de lecture de la taille du problème\n");
if (n_it == 0 || n_op == 0)
        errx(1, "Impossible d'avoir 0 objets ou 0 options");
instance->n_items = n_it;
instance->n_options = n_op;
instance->item_name = malloc(n_it * sizeof(char *));
instance->ptr = malloc((n_op + 1) * sizeof(int));
instance->options = malloc(n_it * n_op *sizeof(int));         // surallocation massive
if (instance->item_name == NULL || instance->ptr == NULL || instance->options == NULL)
        err(1, "Impossible d'allouer la mémoire pour stocker la matrice");
```

On aborde maintenant la partie plus compliquée : lire des lignes de taille arbitraire, et des identifiants de taille arbitraire. On lit les octets un par un. La lecture fonctionne donc avec un automate fini à six états : début d'une ligne, lecture d'un identifiant, lecture d'un espace, lecture de `|`, fin de la ligne, fin du fichier.

25b ⟨*Gestion des I/O* 25b⟩≡ (24) 25c▷

```
enum state_t {START, ID, WHITESPACE, BAR, ENDLINE, ENDFILE};
enum state_t state = START;
```

Pour obtenir les octets un par un, on lit des paquets de taille fixe dans un `buffer` et on examine les octets lus un par un. Si on arrive au bout du buffer, on le re-remplit.

25c ⟨*Gestion des I/O* 25b⟩+≡ (24) ◁25b 26a▷

```
char buffer[256];
int i = 0;     // prochain octet disponible du buffer
int n = 0;     // dernier octet disponible du buffer
```

25d ⟨*Re-remplit le buffer* 25d⟩≡ (26b 28)

```
n = fread(buffer, 1, 256, in);
if (n == 0) {
        if (feof(in)) {
                state = ENDFILE;
        }
        if (ferror(in))
                err(1, "erreur lors de la lecture de %s", in_filename);
}
i = 0;
```

Quand on lit le nom d'un objet, on le copie dans la variable `id`. On impose que les identifiants fassent moins de 64 caractères.

26a ⟨*Gestion des I/O* 25b⟩+≡ (24) ◁25c

```
char id[65];
id[64] = 0;    // sentinelle à la fin, quoi qu'il arrive
int j = 0;     // longueur de l'identifiant en cours de lecture
```

Pour distinguer les espaces du reste, on utilise la fonction `isspace` offert par la librairie C standard.

26b ⟨*Lit les noms des objets* 26b⟩≡ (24)

```
int current_item = 0;
while (state != ENDLINE) {
        enum state_t prev_state = state;
        if (i >= n) {
                ⟨Re-remplit le buffer 25d⟩
        }
        ⟨Détermine le nouvel état 27a⟩
        // traite le caractère lu
        if (state == ID) {
                if (j == 64)
                        errx(1, "nom d'objet trop long : %s", id);
                id[j] = buffer[i];
                j++;
        }
        if (prev_state == ID && state != ID) {
                ⟨Stocke l'objet qui vient d'être lu 27b⟩
        }
        if (state == BAR)
                instance->n_primary = current_item;
        if (state == ENDFILE)
                errx(1, "Fin de fichier prématurée");
        // passe au prochain caractère
        i++;
}
if (current_item != instance->n_items)
        errx(1, "Incohérence : %d objets attendus mais seulement %d fournis\n",
                        instance->n_items, current_item);
if (instance->n_primary == 0)
        instance->n_primary = instance->n_items;
```

27a ⟨*Détermine le nouvel état* 27a⟩≡ (26b 28)

```
if (state == ENDFILE) {
        // don't examine buffer[i]
} else if (buffer[i] == '\n') {
        state = ENDLINE;
} else if (buffer[i] == '|') {
        state = BAR;
} else if (isspace(buffer[i])) {
        state = WHITESPACE;
} else {
        state = ID;
}
```

27b ⟨*Stocke l'objet qui vient d'être lu* 27b⟩≡ (26b)

```
id[j] = '\0';
if (current_item == instance->n_items)
        errx(1, "Objet excedentaire : %s", id);
for (int k = 0; k < current_item; k++)
        if (strcmp(id, instance->item_name[k]) == 0)
                errx(1, "Nom d'objets dupliqué : %s", id);
instance->item_name[current_item] = malloc(j+1);
strcpy(instance->item_name[current_item], id);
current_item++;
j = 0;
```

Le traitement des options est assez similaire. Comme on a suralloué la mémoire, on n'est pas obligé de gérer la taille des options.

28 ⟨*Lit les options* 28⟩≡ (24)

```
int current_option = 0;
int p = 0;       // pointeur courant dans instance->options
instance->ptr[0] = p;
bool has_primary = false;
while (state != ENDFILE) {
        enum state_t prev_state = state;
        if (i >= n) {
                ⟨Re-remplit le buffer 25d⟩
        }
        ⟨Détermine le nouvel état 27a⟩
        // traite le caractère lu
        if (state == ID) {
                if (j == 64)
                        errx(1, "nom d'objet trop long : %s", id);
                id[j] = buffer[i];
                j++;
        }
        if (prev_state == ID && state != ID) {
                ⟨Ajoute l'objet qui vient d'être lu à l'option en cours de lecture 29a⟩
        }
        if (state == BAR) {
                errx(1, "Trouvé | dans une option.");
        }
        if ((state == ENDLINE || state == ENDFILE)) {
                // esquive les lignes vides
                if (p > instance->ptr[current_option]) {
                        ⟨Stocke l'option qui vient d'être lue 29b⟩
                }
        }
        // passe au prochain caractère
        i++;
}
if (current_option != instance->n_options)
        errx(1, "Incohérence : %d options attendues mais seulement %d fournies\n",
                        instance->n_options, current_option);
```

29a ⟨*Ajoute l'objet qui vient d'être lu à l'option en cours de lecture* 29a⟩≡ (28)

```
id[j] = '\0';
// identifie le numéro de l'objet en question
int item_number = -1;
for (int k = 0; k < instance->n_items; k++)
        if (strcmp(id, instance->item_name[k]) == 0) {
                item_number = k;
                break;
        }
if (item_number == -1)
        errx(1, "Objet %s inconnu dans l'option #%d", id, current_option);
// détecte les objets répétés
for (int k = instance->ptr[current_option]; k < p; k++)
        if (item_number == instance->options[k])
                errx(1, "Objet %s répété dans l'option %d\n",
                                instance->item_name[item_number], current_option);
instance->options[p] = item_number;
p++;
has_primary |= item_is_primary(instance, item_number);
j = 0;
```

29b ⟨*Stocke l'option qui vient d'être lue* 29b⟩≡ (28)

```
if (current_option == instance->n_options)
        errx(1, "Option excédentaire");
if (!has_primary)
        errx(1, "Option %d sans objet primaire\n", current_option);
current_option++;
instance->ptr[current_option] = p;
has_primary = false;
```

Références

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[Knu92] Donald E. Knuth. *Literate programming*, volume 27 of *CSLI lecture notes series*. Center for the Study of Language and Information, 1992.

[Knu19] Donald E. Knuth. *Dancing Links*, volume 4B, Fascicle 5 of *The Art of Computer Programming*. Addison-Wesley, 2019. Available online at `https://www-cs-faculty.stanford.edu/~knuth/fasc5c.ps.gz`.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5) :97–105, 1994.