Chapitre 7

Retour sur trace

Notions	Commentaires
Recherche par force brute. Retour sur trace (*Backtracking*).	On peut évoquer l'intérêt d'ordonner les données avant de les parcourir (par exemple par une droite de balayage).

Extrait de la section 4.2 du programme officiel de MP2I : « Exploration exhaustive ».

Sommaire

On s'intéresse dans ce chapitre à la recherche de l'existence d'une solution parmi *plein* de possibilités.

0 Fils rouges

0.0 Sudoku

Le Sudoku est un casse-tête qui consiste à remplir une grille 9×9 en faisant en sorte que dans chaque ligne, colonne, et gros carré il y ait chaque chiffre de $[\![1; 9[\![$ exactement une fois.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Une grille de Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(b) Une grille de Sudoku résolue.

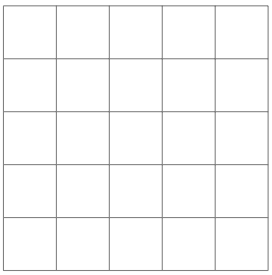
FIGURE 7.1 – Grille de Sudoku. Les gros carré ont des bords en gras.

Remarque. Ceci est le 9-Sudoku, mais on peut aussi définir des Sudoku n'importe quel côté de longueur carrée : 4, (9), 16, 25, etc.

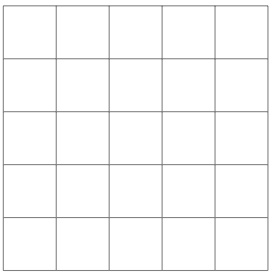
0.1 *n* dames

Un autre problème classique est le problème des n dames :

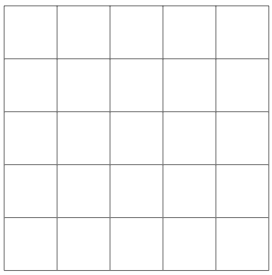
- ENTRÉE : n un entier strictement positif.
- TÂCHE : Trouver s'il est possible et comment placer n dames sur un échiquier $n \times n$ sans que deux dames ne soient en prise (deux dames sont en prises si elles sont alignées verticalement ou horizontalement ou diagonalement).



(a) Une prise horizontale



(b) Une prise verticale



(c) Une prise diagonale

FIGURE 7.2 – Exemples des 3 natures de prises possibles sur échiquier 5×5

1 Exploration exhaustive

1.0 Définition et limites

Définition 1 (Exploration exhaustive).
L'**exploration exhaustive** consiste à parcourir toutes les potentielles solutions d'un problème, et à vérifier les unes après les autres s'il s'agit bel et bien de solutions ou non.
On parle aussi de **force brute** (« brute force » en VO) : on essaye pas de faire un algorithme intelligent ; on utilise simplement et stupidement la force de calcul de notre ordinateur.

Exemple.

- Générer toutes les façons de parcourir une grille de Sudoku (il y en a $9^{nombre\_de\_cases\_vides}$), puis pour chacune de ces façons tester si elle est valide ou non.
  C'est *très* lent : une grille a $9 \times 9 = 81$ cases. Mettons que la grille soit simple et soit déjà à moitié remplie : cela fait $9^{40} \simeq 10^{38}$ possibilités. Sur un processeur de ~1GHz, il faut $\sim 10^{29}$ secondes soit $\sim 4.10^{21}$ ans. C'est dix millions de fois l'âge de l'univers.
- Pour les n dames, il y a n dames à placer sur un damier à n^2 cases, donc n^{2^n} possibilités. Avec $n = 8$ et un processeur à ~1GHz, il faut 78 heures. C'est plus raisonnable que le Sudoku, mais cela reste beaucoup.

Remarque. Le terme « force brute » est un terme général en informatique, qui désigne souvent mais pas toujours l'exploration exhaustive. L'idée des méthodes par fruce brute est cependant toujours la même : ne pas réflechir et faire (aveuglément) confiance au GHz.

Proposition 2 (Inefficacité des explorations exhaustives).
Les explorations exhaustives (et plus généralement toutes les méthodes par force brute) ne fonctionnent que sur des petites entrées.
Leur principal avantage est d'être simples à coder, et de permettre de vérifier les résultats renvoyés par des algorithmes plus avancés.

Il faut pour cela découper une potentielle solution en plusieurs « choix » successifs. Ainsi, au lieu

Exemple.

- Pour le Sudoku, il faut d'abord choisir ce que l'on met dans la première case, puis choisir ce que l'on met dans la seconde, etc.
- Pour les n dames, il faut d'abord choisir où placer la première dame, puis où placer la seconde, etc.

Définition 3 (Vocabulaire).
On considère un problème dont le but est de déterminer s'il existe un uplet $(x_0, ..., x_{\ell-1})$ vérifiant une certaine propriété. Pour construire un tel uplet, on choisit d'abord x_0, puis x_1, etc.
Dans ce cours, on appelle :

- **solution partielle** : c'est « une solution en cours de construction où il reste des choxi à faire ». Formellement, c'est un uplet $(x_0, x_1, ..., x_{i-1})$ avec $i - 1 < \ell$.
- **solution complète** : c'est « une solution entièrement remplie, où il n'y a plus de choix à faire ». Formellement, c'est un uplet $(x_0, ..., x_{\ell-1})$.
- **solution valide**, ou juste **solution** : c'est une solution complète qui vérifie la propriété attendue.

De plus, si s est une solution partielle et s' une solution partielle ou complète, on dit que s' **est une extension de** s lorsque « s' a été obtenue à partir de s en faisant des choix supplémentaires ». Formellement, si $s = (x_0, x_1, ..., x_{i-1})$ et $s' = (y_0, y_1, ..., y_{i-1})$, s' étend s si et seulement si $j \geq i$ et $\forall k \in [\![0; i[\![, x_k = y_k$.

Exemple.

- Une solution partielle du Sudoku est une grille partiellement remplie. Une solution complète une grille entièrement remplie, et une valide une grille entièrement remplie qui respecte les règles.
- Une solution partielle des n dames est le placement de i dames. Une solution complète est le placement de n dames, et une valide est n dames qui ne sont pas en prise.

Remarque.

- Cette formalisation n'est pas parfaite (par exemple, elle sous-entend que toutes les solutions complètes ont la même longueur ce qui peut-être discutable), mais a l'avantage de nous donner du vocabulaire et des notations que je peux réutiliser dans tout ce cours.
- Ce n'est pas du vocabulaire « officiel », redéfinissez ces termes avant de les utiliser si vous en avez besoin en concours.

1.1 Pseudo-code général d'une exploration exhaustive

Pour coder une exploration exhaustive, la récursivité est très souvent utile : on fait un choix, puis les choix suivants sont faits récursivement.

Il faut de plus disposer d'un vérificateur, une fonction qui vérifie si une solution potentielle est bel et bien une solution ou non. On peut alors appliquer l'algorithme récursif ci-dessous :

Algorithme 10 : EXHAUSTIF

Entrées : $sol = (x_0, ..., x_{i-1})$ une solution partielle ou complète
Sorties : Vrai si et seulement si *sol* peut-être étendue en une solution valide.
Effet secondaire : afficher une solution valide si elle existe.

```
si sol est complète alors
    si VÉRIFICATEUR(sol) alors
        Afficher sol
        renvoyer Vrai
    sinon
        renvoyer Faux

// Sinon : tester toutes les extensions possibles
pour chaque valeur x_i possible pour le prochain choix faire
    // Essayer avec ce x_i
    sol ⟵ sol étendue avec x_i
    si EXHAUSTIF(sol) alors
        renvoyer Vrai

    // Sinon, annuler le choix x_i et en tester un autre au prochain tour de boucle
    sol ⟵ sol sans x_i

// Si aucun choix n'a fonctionné, on renvoie Faux
renvoyer Faux
```

Remarque.

- Le point très important de cet algorithme est qu'il faut annuler un choix s'il n'a pas fonctionné ! En effet, si *sol* est mutable (comme dans ce pseudo-code), ne pas annuler les choix mène à des erreurs du type « un appel récursif a fait un mauvais choix, n'a pas enlevé ce choix, donc les appels récursifs suivants qui testent d'autres choix ont toujours ce mauvais choix à l'intérieur de leur *sol* et plantent ».
- Des écritures non-récursives sont possibles mais généralement plus techniques à rédiger. On y utilise parfois des compteurs n-aires pour représenter et parcourir des sous-ensembles.

1.2 Exploration exhaustive des n dames

Voici des extraits pertinents de `exhaustif.c` qui implémente la méthode générale présentée ci-dessus :

exhaustif.c

```c
/*
Choix d'implémentation :
- le plateau est indicé comme une matrice n*n (avec (0,0) en haut à gauche)
- les n dames sont stockées dans un tableau dames.
- une dame absente correspond aux coordonnées (-1,-1)
*/
```

ndames.h

```c
/** Une coordonnée est une paire (ligne, colonne) */
struct coo_s {
  int lgn;
  int col;
};
typedef struct coo_s coo;
```

Et voici le retour sur trace (sans l'affichage de la solution) :

exhaustif.c

```c
/** Renvoie true ssi il existe une façon de compléter
  * les nb_dames déjà placées qui respecte les règles.
  *
  * dames est le tableau des nb_dames placées,
  * n le nombre de dames à placer.
  */
bool rec_exhaustif(coo* dames, int nb_dames, int n) {
  if (nb_dames == n) {
    return verificateur(dames, n);
  }

  // Parcourir toutes les façons de placer une dame
  for (int lgn = 0; lgn < n; lgn += 1) {
    for (int col = 0; col < n; col += 1) {
      dames[nb_dames].lgn = lgn;
      dames[nb_dames].col = col;
      // Si mettre cette dame mène à une solution : GG
      if (rec_exhaustif(dames, nb_dames+1, n)) {
        return true;
      }
      // Sinon, enlever cette dame
      dames[nb_dames].lgn = -1;
      dames[nb_dames].col = -1;
    }
  }

  return false;
}
```

Figure 7.3 – (Partie de l') arbre des appels récursifs qui ont lieu pour $n = 3$

1.3 Amélioration de l'exploration exhaustive des n dames

Sans pour autant faire un algorithme très intelligent, on peut proposer une amélioration de la fonction précédente : d'après la règle de la prise verticale, il y a au plus une dame par colonne. On peut donc construire une solution ainsi :

- Choisir la ligne de la dame de la colonne 0.
- Puis choisir la ligne de la dame de la colonne 1.
- Etc.

On obtient le code `exhaustif-ameliore.c` dont voici des extraits pertinents :

exhaustif-ameliore.c

```c
/*
Choix d'implémentation :
- le plateau est indicé comme une matrice n*n (avec (0,0) en haut à gauche)
- la ième dame est en colonne i.
- le tableau dames_lgn stocke les lignes des dames
- une dame absente correspond à la ligne -1
- notez que le type coo ne sert plus
*/
```

exhaustif-ameliore.c

```c
/** Renvoie true ssi il existe une façon de compléter
  * les nb_dames déjà placées qui respecte les règles.
  *
  * dames_lgn est le tableau qui à l'indice d'une dame
  * placée associe sa colonne (la dame i est en ligne i),
  * nb_dames le nombre de dames déjà placées,
  * n le nombre de dames à placer.
  */
bool rec_exhaustif(int* dames_lgn, int nb_dames, int n) {
  if (nb_dames == n) {
    return verificateur(dames_lgn, n);
  }

  // Parcourir toutes les lignes pour la prochaine dame
  for (int lgn = 0; lgn < n; lgn += 1) {
    dames_lgn[nb_dames] = lgn;
    if (rec_exhaustif(dames_lgn, nb_dames+1, n)) {
      return true;
    }
    dames_lgn[nb_dames] = -1; // ne pas oublier d'annuler le choix !!
  }

  return false;
}
```

Remarquez qu'on a une boucle `for` imbriquée de moins ! Le gain en temps n'est pas négligeable :

Algorithme employé	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
Exhaustif	2ms	4ms	1,2s	26s	3 heures
Exhaustif amélioré	2ms	1ms	2ms	2ms	25ms

FIGURE 7.4 – Durées d'exécution sur un processeur à ~3GHz

Remarque.

- Le gain de temps spectaculaire de cette simple amélioration s'explique par le fait qu'ajoute la condition « une dame par colonne » réduit énormément les possibilités (passage à la racine !) :

Contraintes	Nombre de possibilités pour n dames	$n = 4$	$n = 7$	$n = 8$
Aucune	n^{2n}	65536	$6,8 \cdot 10^{11}$	$2,8 \cdot 10^{14}$
Une dame par case	$\binom{n^2}{n}$	1820	$8,6 \cdot 10^{7}$	$4,4 \cdot 10^{9}$
Une dame par ligne	n^n	256	823543	$1,7 \cdot 10^{7}$
Une dame par ligne et colonne	$n!$	24	5040	40320

FIGURE 7.5 – Nombres de façons de placer n dames sur un échiquier $n \times n$ selon différentes contraintes. Les écritures scientifiques sont données à 2 chiffres significatifs.

- Le 1ms de l'exhaustif amélioré pour $n = 5$ n'est pas une erreur : c'est un « coup de chance », il se trouve que pour $n = 5$ ma recherche récursive trouve une très vite et doit peu souvent « essayer un autre choix ».

  En effet, mes fonctions s'arrêtent dès qu'elles ont trouvé une solution : elles n'explorent donc pas *tout* l'espace des possibilités.

2 Retour sur trace

2.0 Définition

Définition 4 (Rejet et retour sur trace).
Une solution partielle est dite **rejettable** si elle ne peut pas être étendue en une solution valide. Un algorithme de **retour sur trace** est une accélériation de l'exploration exhaustive qui après chaque choix essaye de détecter si la solution partielle obtenue est rejettable. Si oui, on annule ce choix, sinon on essaye récursivement d'étendre cette solution partielle.

Remarque.

- Toute la difficulté du retour sur trace est d'écrire une fonction qui détecte le plus fidèlement possible les solutions rejettables ! Dans les deux exemples fils rouge de ce cours c'est facile, dans d'autres cas plus compliqué.
- Pour accélérer la recherche, en plus d'un bon rejet il est utile d'avoir une bonne « intuition » : si l'on arrive à prévoir quel choix a le plus de chance d'être étendue en une solution valide, on peut tester ce choix en premier (et tester en dernier les choix les moins prometteurs).
- En MPI, vous étudierez les algorithmes *Branch and Bound*. Il s'agit d'une adaptation de la méthode de retour sur trace à l'optimisation.
- Le terme « trace » désigne l'historique d'un programme. Un algorithme par « retour sur trace » est donc un algorithme qui peut revenir en arrière et remettre en cause des choix précédents. Avec notre façon d'écrire le code, ce retour en arrière est très simple et géré par la sortie des appels récursifs.

Exemple.

- Pour le Sudoku, on peut rejetter une solution dès qu'elle a deux chiffres identitiques sur une même ligne / colonne / grand carré. Notez que cela suffira à prouver que la grille n'est pas remplie, car si les 9 chiffres d'une ligne sont distincts alors les 9 chiffres sont présents, et idem pour les lignes et carrés. [1]

1. Pour tout E fini et : $f : E \to E$, f est injective si et seulement si elle est bijective (si et seulement si elle est surjective).

- Pour les n dames, on peut rejetter dès que 2 dames sont sur la même ligne / colonne / diagonale.

Voici par exemple pour $n = 4$ l'exploration que l'on obtient avec ce rejet :

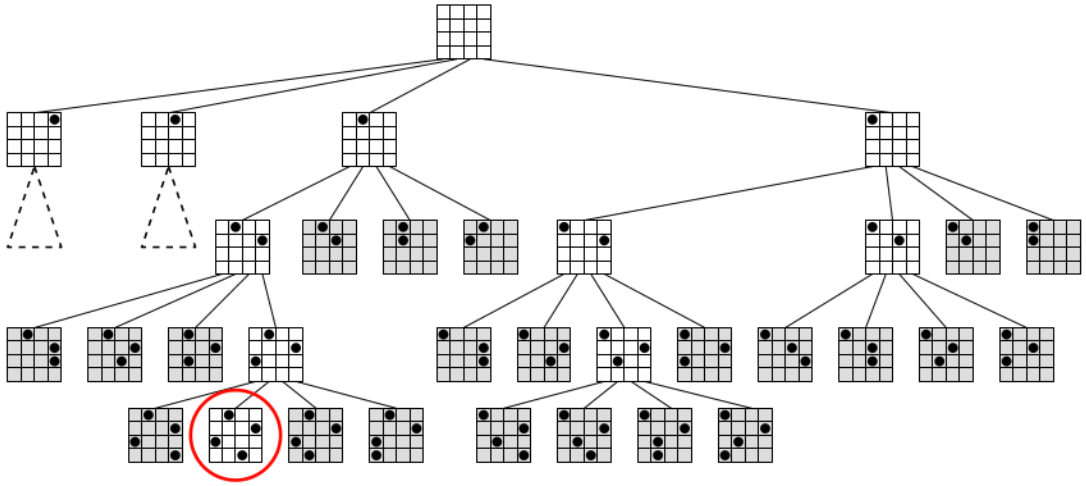


FIGURE 7.6 – Exploration en plaçant d'abord sur la première ligne, puis sur la seconde, etc. En gris les solutions partielles rejetées. La moitié non-représentée de la figure est symétrique. Src : J.B. Bianquis

Le pseudo-code générique d'une recherche exhaustive peut-être modifié ainsi pour obtenir un retour sur trace :

Algorithme 11 : RETOURSURTRACE

Entrées : $sol = (x_0, ..., x_{i-1})$ une solution partielle ou complète
Sorties : Vrai si et seulement si *sol* peut-être étendue en une solution valide.

```
1 si sol est complète alors
2 |   renvoyer VÉRIFICATEUR(sol)

3 pour chaque valeur x_i possible pour le prochain choix faire
  |   // Essayer le choix x_i
4 |   sol ←sol étendue avec x_i
5 |   si REJET(sol) ne rejette pas la solution partielle alors
6 |   |   si RETOURSURTRACE(sol) alors
7 |   |   |   renvoyer Vrai
  |   // Sinon : annuler le choix x_i et en essayant un autre à la prochaine iter
8 |   sol ←sol sans x_i

9 renvoyer Faux
```

⚠ J'insiste : penser à défaire un choix s'il n'a pas fonctionné (lgn 8) est très important et vous évitera énormément de débogage !

2.1 Retour sur trace pour les n dames

On se base sur la verison améliorée de la recherche exhaustive, à laquelle on veut ajouter un rejet. Pour cela, après chaque choix on va tester si la dame que l'on vient d'ajouter est en conflit avec une des dames précédentes. La fonction `check_dame` réalise cela. On obtient le code suivant pour le retour sur trace :

backtrack.c

```c
/** Renvoie true ssi il existe une façon de compléter
  * les nb_dames déjà placées qui respecte les règles.
  *
  * dames_lgn est le tableau qui à l'indice d'une dame
  * placée associe sa colonne (la dame i est en ligne i),
  * nb_dames le nombre de dames déjà placées,
  * n le nombre de dames à placer.
  */
bool backtrack(int* dames_lgn, int nb_dames, int n) {
  if (nb_dames == n) {
    return verificateur(dames_lgn, n);
  }

  for (int lgn = 0; lgn < n; lgn += 1) {
    dames_lgn[nb_dames] = lgn;
    if (check_dame(dames_lgn, nb_dames)
        && backtrack(dames_lgn, nb_dames+1, n) )
    {
      return true;
    }

    dames_lgn[nb_dames] = -1;
  }

  return false;
}
```

Remarque.

- Notez l'usage de l'évaluation paresseuse du `&&` en lignes 97-98 : l'exploration récursive n'est effectuée *que* si la solution n'est pas rejetée.
- Notez la très grande similarité avec la recherche exhaustive. En fait, la recherche exhaustive est un cas particulier de retour sur trace : celui où notre fonction rejet est très très très mauvaise.
- La ligne 92 peut-être simplifiée : si l'on atteint cette ligne, c'est que l'on a réussi à placer n dames sans qu'aucune d'entre elles ne soit en prise avec une dame d'avant. Autrement dit, aucune dame n'est en prise : on peut `return true` .
- Cette fonciton est *très* performante : 2ms pour $n = 8$, 72ms pour $n = 20$.

2.2 Variantes

Jusque là, le problème que nous résolvions était de la forme « existe-t-il une solution » ? Il existe en fait 4 variantes classiques pour un retour sur trace :

- Déterminer l'existence d'une solution valide.
- Renvoyer une solution valide si elle existe.
- Trouver le nombre de solutions valides.
- Renvoyer toutes les solutions valides.

Pour les variantes où il faut renvoyer une (resp. ou des) solutions, le plus simple est de faire en sorte que les appels récursifs se partagent l'adresse d'un endroit où aller recopier (resp. ajouter une copie de) une solution valide quand on en trouve une.

Attention à bien *recopier* la solution trouvée, cela évite des erreurs du type « la solution valide que j'ai trouvée a été modifiée par les appels récursifs suivants et n'est plus valide ».

Remarque. Tout ceci est très général, les choix précis à faire dépendent du langage et des choix d'implémentation.

3 Complexité

3.0 Analyse théorique

Il s'agit de calculer la complexité d'un algorithme récursif. Avec les retours sur trace, le plus simple est généralement de :

- Majorer grossièrement le coût de la fonction de rejet.
- De même pour la vérification, si elle est distincte du rejet.
- Majorer le nombre d'appels qui ont lieu par le nombre d'appels qui auraient lieu sans aucun fonction de rejet. Cela est généralement assez simple car il s'agit du nombre de noeuds d'un arbre parfait.
- Multiplier le coût des rejets et vérifications par le nombre d'appels. Par exemple

Exemple. Pour les n dames, le coût réel du rejet est $O(nombres\_de\_dames\_placees)$. On peut majorer cela par $O(n)$ pour simplifier. La vérification n'est pas nécessaire (cf remarque à propos de la ligne 92) et on l'enlève donc. Chaque appel fait au plus n appels récursifs (les n façons de placer la prochaine dame), et la profondeur de l'arbre d'appels est de n. L'arbre d'appels contient donc n^n appels. Ainsi, au total, une borne supérieure de la complexité de ce code est $O(n^{n+1})$.

Proposition 5 (Limites de l'analyse théorique).
Sur les retours sur trace, l'analyse théorique présentée ci-dessus est très limitée. Le bon outil d'analyse est la mesure du temps d'exécution en pratique.
En effet, tout l'intérêt d'un retour sur trace est de ne *pas* explorer toutes les possibilités, et beaucoup beaucoup beaucoup de solutions partielles sont rejetées.

3.1 Importance de l'ordre des appels

En plus d'un bon rejet, un point important pour accélérer un retour sur trace est de tester les appels récursifs dans le bon ordre : on veut faire en premier les choix les plus probables !

Exemple. Sur le Sudoku, on peut choisir le contenu des cases dans l'ordre, de gauche à droite et de haut en bas. Cela fonctionne, mais n'est pas le plus efficace. Par exemple, sur la grille ci-dessous, on voit que certaines cases ont 4 valeurs qui peuvent y être mises sans rejet immédiat, alors que d'autres n'en ont qu'1. Il vaut mieux compléter en premier ces dernières (pour se rapprocher plus vite d'une solution valide ou pour rejeter plus vite) !

1234	23	123	2
123	23	123	4
23	4	2	1
2	1	24	3

FIGURE 7.7 – Valeurs non-immédiatement rejetables (en rouge) pour les cases vides d'une grille de 4-Sudoku (dont les chiffres déjà placés sont en noir)

(a) Un fil avec deux séquences adjacentes identiques

(b) Un autre fil avec deux séquences adjacentes identiques

(c) Un fil avec 10 perles sans séquences adjacentes identiques

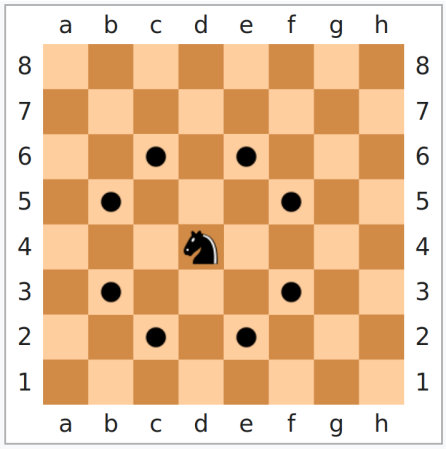
4 Autres exemples classiques

Voici d'autres exemples de retour sur trace classiques, que nous ferons en TP ou en DM[2].

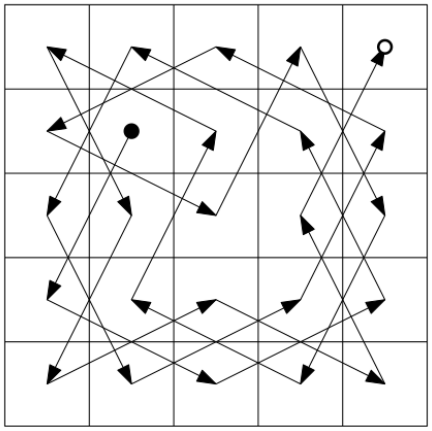
4.0 Cavalier d'Euler

Le problème du cavalier d'Euler est le suivant :

- Entrée : n la dimension du côté d'un échiquier et (i, j) les coordonnées d'une case de l'échiquier
- Tâche : déterminer si un cavalier initialement placé en (i, j) peut se déplacer sur l'échiquier de sorte à se poser sur toutes les cases mais jamais deux fois sur la même.



(a) Les 8 mouvements possibles d'un cavalier



(b) Résolution du cavalier d'Euler sur un échiquier 5×5 en partant de $(1, 1)$

FIGURE 7.8 – Autour du cavalier d'Euler

4.1 Perles de Dijkstra

Les perles de Dijkstra est un exercice historique de programmation :

- Entrée : n un entier strictement positif
- Tâche : on dispose de perles Bleu, Blanche et Rouge[3]. Comment en enfiler n sur un fil de sorte à ce qu'il n'y ait pas deux séquences adjacentes identiques ?

D'un point de vue plus mathématique, il s'agit de rechercher un mot ternaire sans carré. Pour en savoir plus : `https://fr.wikipedia.org/wiki/Mot_sans_facteur_carr%C3%A9` .

2. Le Sudoku sera un DM !
3. Comme le drapeau néerlandais, Dijkstra étant néerlandais. Vous pensiez à quel pays ?

4.2 SubsetSum

Le problème de la somme d'un sous-ensemble, aussi appelé SubsetSum, est le suivant :

- Entrée : $E \subset \mathbb{N}$ un ensemble fini d'entiers positifs et $t \in \mathbb{Z}$ une « cible » (**t**arget).
- Tâche : trouver $P \subseteq E$ tel que $\sum_{x \in P} x = t$

C'est un problème très célèbre en informatique, car c'est un exemple classique de problème NP-complet. Rendez-vous en MPI pour en savoir plus !

4.3 CNF-SAT

Dans le cours de Logique, nous verrons le problème CNF-SAT qui consiste à cherche une manière de satisfaire une formule logique. Nous y verrons l'algorithme de Quine, qui est un retour sur trace !

5 Écriture itérative

Il existe des façons impératives d'écrire les retours sur trace. **Je les décommande fortement, car il est bien plus simple de faire des erreurs ainsi.**

Il demande de pouvoir ordonner les valeurs possibles pour le i-ème choix. Étant donnée x_i une valeur possible pour le $i$ème choix, on note Suivant(x_i) la valeur à tester pour le ième choix si x_i n'a pas permis d'obtenir une solution valide. Quand il n'y a plus de prochaine valeur possible pour ce choix, Suivant renvoie une erreur ou une valeur particulière. On note aussi Premier(i) le premier choxi à tester pour le *ime* choix.

Exemple.

- pour le Sudoku, le i-ème choix est le contenu de la $i$ème case, et on peut ordonner les valeurs possibles pour ce choix en posant $0 < 1 < ... < 9$. Donc Suivant(1) = 2, etc.
- pour les n dames (avec une dame par colonne), le $i$ème choix est la ligne de la $i$ème dame. On les ordonne là aussi naturellement : $0 < ... < n - 1$.

Algorithme 11 : RetourSurTrace impératif (à éviter !!)

Entrées : ℓ la longueur de solution valide voulue
Sorties : Vrai si et seulement si il existe une solution valide de longueur ℓ

```
1  sol ←tableau à ℓ cases // solution partielle
2  len ←0 // longueur de la solution partielle
   // boucle "infinie" dont on devrait finir par sortir
3  tant que Vrai faire
4      x ←Suivant(sol[len − 1])
5      si len = ℓ alors
6          si x est une erreur alors
7              renvoyer Faux
8          sinon
9              sol[len − 1] ←x
10             si Vérificateur(sol) alors renvoyer Vrai
               // Sinon : tour de boucle suivant
11     sinon
           // Sinon : len < ℓ
12         si x est une erreur alors
13             len ← len − 1
14         sinon
15             sol[len − 1] ←x
16             si non(Rejet(sol, len)) alors
17                 sol[len] ←Premier(len)
18                 len ← len + 1
               // Sinon : tour de boucle suivant
```