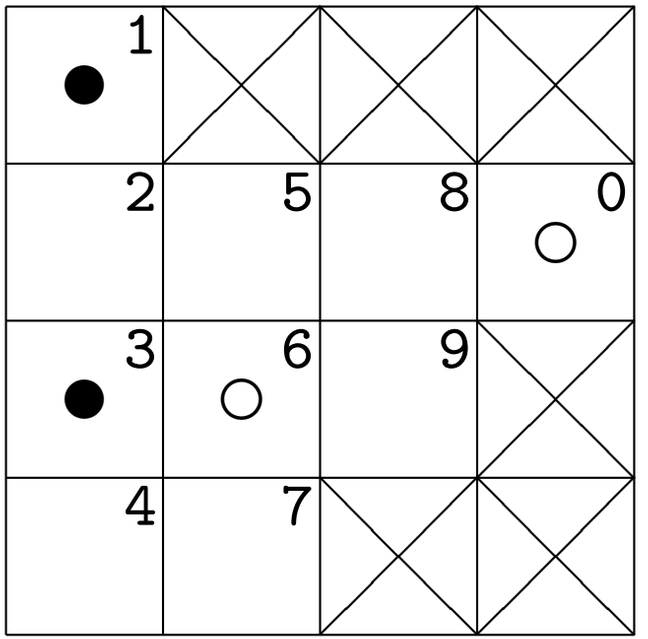
Problème des Cavaliers I

Comment permuter les Cavaliers,
les cases barrées étant interdites ?



Configuration `(a b c d)` :

le Cavalier noir se trouvant initialement en `1` se trouve en `a` ;
le Cavalier noir se trouvant initialement en `3` se trouve en `b` ;
le Cavalier blanc se trouvant initialement en `6` se trouve en `c` ;
le Cavalier blanc se trouvant initialement en `0` se trouve en `d`.

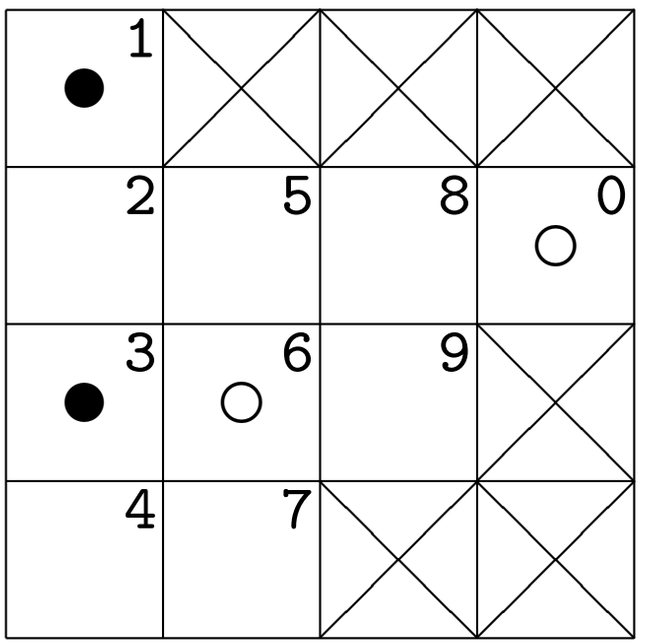
Mouvement `(a . b)` : de la position `a` vers la position `b`.

Problème des Cavaliers II

Variables globales : situation initiale, liste des mouvements possibles :

```
(define *init* '(1 3 6 0))


(define *moves*
  '((1 . 6) (1 . 8) (2 . 7) (2 . 9)
    (3 . 8) (4 . 5) (4 . 9) (5 . 4)
    (6 . 1) (6 . 0) (7 . 2) (7 . 8)
    (8 . 1) (8 . 3) (8 . 7) (9 . 2)
    (9 . 4) (0 . 6)))
```



On voit par exemple qu'un Cavalier se trouvant en position 8 peut aller en un coup à la position 1, 3 ou 7 (si cette position est libre).

Problème des Cavaliers III

Les situations finales acceptables sont peu nombreuses ; en effet, la liste `(1 3 6 0)` admet $4! = 24$ permutations, dont 9 sont des dérangements (permutations sans point fixe).

```
(define *final* '((0 1 3 6) (0 6 1 3) (0 6 3 1)
                  (3 0 1 6) (3 1 0 6) (3 6 0 1)
                  (6 0 1 3) (6 0 3 1) (6 1 0 3)))
```

Si on se limite aux permutations dans lesquelles les Cavaliers blancs prennent la place des Cavaliers noirs (et réciproquement), on a :

```
(define *best* '((0 6 1 3) (0 6 3 1) (6 0 1 3) (6 0 3 1)))
```

```
(define n= (lambda (x y) (not (= x y))))  ;; case libre?
(define k1 car)   (define k2 cadr)  (define k3 caddr)  (define k4 cadddr)
(define org car)  (define dst cdr)        ;; accesseurs
```

Problème des Cavaliers IV

La fonction centrale du problème est la fonction `succ`. Elle prend comme arguments une situation et un mouvement ; si ce mouvement est possible dans la situation donnée, la fonction renvoie la situation résultante, sinon elle renvoie `#f`. On a :

```
(define succ
  (lambda (sit mv)
    (let ((a (k1 sit)) (b (k2 sit)) (c (k3 sit)) (d (k4 sit))
          (e (org mv)) (f (dst mv)))
      (cond
        ((and (= a e) (n= b f) (n= c f) (n= d f)) (list f b c d))
        ((and (= b e) (n= a f) (n= c f) (n= d f)) (list a f c d))
        ((and (= c e) (n= a f) (n= b f) (n= d f)) (list a b f d))
        ((and (= d e) (n= a f) (n= b f) (n= c f)) (list a b c f))
        (else #f)))))
```

Les cinq clauses de la forme `cond` correspondent aux cinq cas possibles : l'origine du mouvement correspond à l'une des quatre positions de la situation donnée ou à aucune.

Problème des Cavaliers V

Sur base de la fonction `succ`, on peut définir une fonction `succs` prenant comme arguments une situation et une liste de mouvements et renvoyant la liste triée (ordre lexicographique) et sans répétitions des situations résultantes.

```
(define succs
  (lambda (sit mvs)
    (if (null? mvs)
        '()
        (let ((s1 (succ sit (car mvs))) (s (succs sit (cdr mvs))))
          (if s1 (insert-sit s1 s) s)))))
(define insert-sit (lambda (sit sits) (add-elem sit sits (lex < =))))
(succs *init* *moves*) ((1 8 6 0) (8 3 6 0))
```

Une situation σ appartient à [[`(succs sit mvs)`]] s'il existe un mouvement μ appartenant à [[`mvs`]] qui permette de passer de [[`sit`]] à σ.
La condition d'un `if` est assimilée à vrai dès qu'elle n'est pas fausse ; c'est pourquoi il était intéressant que la fonction précédente `succ` renvoie `#f` pour signaler l'absence de situation résultante.

Problème des Cavaliers VI

```
(define succss
  (lambda (sits mvs)
    (if (null? sits)
        '()
        (let ((l1 (succs (car sits) mvs)) (l (succss (cdr sits) mvs)))
          (merge-sits l1 l)))))

(define merge-sits          ;; fusion de
  (lambda (s1 s2)           ;; deux listes triées
    (cond ((null? s1) s2)
          ((null? s2) s1)
          ((equal? (car s1) (car s2))
           (cons (car s1) (merge-sits (cdr s1) (cdr s2))))
          (((lex < =) (car s1) (car s2))
           (cons (car s1) (merge-sits (cdr s1) s2)))
          (else (cons (car s2) (merge-sits s1 (cdr s2)))))))
```

Une situation σ appartient à [[`(succs sits mvs)`]] s'il existe une situation ρ appartenant à [[`sits`]] et un mouvement μ appartenant à [[`mvs`]] qui permette de passer de ρ à σ.

Problème des Cavaliers VII

La fonction `succss` permet en principe de résoudre le problème posé.
En effet, par applications successives, on peut construire l'ensemble des situations accessibles depuis la situation initiale en un coup, en deux coups, etc. Cette approche naïve est peu efficace car elle ne tient pas compte des cycles et, comme tout mouvement est réversible, les cycles sont nombreux.

On définit donc une fonction `new-succss` qui ne donne que les "nouveaux" successeurs, c'est-à-dire les situations non encore rencontrées.
Les situations déjà rencontrées, et donc dorénavant interdites, sont groupées dans un troisième argument. On a :

```
(define new-succss
  (lambda (sits mvs forbid)
    (let ((ss (succss sits mvs)))
      (diff ss forbid))))          ;; diff: différence ensembliste
```

$\sigma \in$ [[`(new-succs sits mvs forbid)`]] si et seulement si
$\sigma \in$ [[`(succs sits mvs)`]] et $\sigma \notin$ [[`forbid`]].

Problème des Cavaliers VIII

```
(define gen                              ;; RESOUT LE PROBLEME
  (lambda (n inits finals)
    (if (= n 0)
        (list 0 '() inits 0 1 '())
        (let* ((rec1 (gen (- n 1) inits finals))
               (rec (cdr rec1)))
          (let ((old (car rec)) (new (cadr rec))
                (lold (caddr rec)) (lnew (cadddr rec)))
            (let ((old1 (merge old new))
                  (new1 (new-succss new *moves* old)))
              (let ((lold1 (+ lold lnew))
                    (lnew1 (length new1))
                    (int (inter new1 finals)))
                (newline)
                (display (list n lold1 lnew1 int))
                (list n old1 new1 lold1 lnew1 int)))))))))
```

Résultats :

`n`	nombre de coups
`old1`	situations accessibles en moins de `n` coups
`new1`	situations accessibles en `n` coups mais pas moins
`lold1`	longueur de `old1`
`lnew1`	longueur de `new1`
`int`	situations finales accessibles en `n` coups mais pas moins

Problème des Cavaliers IX

La fonction `gen` imprime des résultats intermédiaires intéressants.

```
(gen 4 (list *init*) *final*)
    (1 1 2 ())   ;; premier coup, deux nouvelles situations
    (2 3 3 ())   ;; deuxième coup, trois nouvelles situations
    (3 6 6 ())   ;; troisième coup, six nouvelles situations
    (4 12 11 ()) ;; quatrième coup, onze nouvelles situations
```

```
(4
 ((1 2 6 0) (1 3 6 0) (1 7 6 0) (1 8 6 0) (2 3 6 0) (7 3 1 0)
  (7 3 6 0) (7 8 6 0) (8 3 1 0) (8 3 1 6) (8 3 6 0) (8 7 6 0))
 ((1 9 6 0) (2 3 1 0) (2 8 6 0) (3 7 6 0) (7 1 6 0) (7 3 1 6)
  (7 3 8 0) (7 8 1 0) (8 2 6 0) (8 7 1 0) (9 3 6 0))
 12   ;; 12 situations accessibles en moins de quatre coups.
 11   ;; 11 situations accessibles en quatre coups mais pas moins,
 ())  ;;    dont aucune n'est finale.
```

Problème des Cavaliers X

La valeur renvoyée est une liste de six résultats. Le premier rappelle que quatre coups consécutifs ont été joués. Le second résultat et le quatrième donnent la liste des situations accessibles en moins de quatre coups et leur nombre (12). Le troisième résultat et le cinquième donnent la liste des situations accessibles en quatre coups mais pas moins, et leur nombre (11). Le sixième résultat est la liste des situations finales accessibles en quatre coups mais pas moins ; on constate que cette liste est vide. De plus, en cours d'exécution, des résultats s'affichent pour un, deux, trois et quatre coups ; ces résultats sont élagués : ils comprennent les longueurs des listes de situations générées, mais pas ces listes elles-mêmes, sauf pour la liste des situations finales accessibles.

Remarque. Il faut éviter la construction d'objets trop gros, et pour cela évaluer *a priori* la taille des objets construits. Dans le cas présent, la taille maximale d'une liste de situations est le nombre de quadruplets ordonnés de nombres distincts compris entre 0 et 9 ; ce nombre est $10 * 9 * 8 * 7 = 5\,040$.

Problème des Cavaliers XI

```
(gen 49 (list *init*) *final*)
    (1 1 2 ())
    ...
    (25 2035 191 ())
    (26 2226 171 ((3 0 1 6) (6 1 0 3)))
    (27 2397 145 ())
    (28 2542 128 ((0 1 3 6) (3 1 0 6) (3 6 0 1)))
    (29 2670 120 ())
    ...
    (39 4166 181 ())
    (40 4347 173 ((0 6 1 3) (6 0 1 3) (6 0 3 1)))
    (41 4520 152 ())
    (42 4672 120 ((0 6 3 1)))
    (43 4792 94 ())
    ...
    (49 5037 3 ())
(49  ;; DETERMINE EXPERIMENTALEMENT
 ((0 1 2 3) (0 1 2 4) ... (9 8 7 5) (9 8 7 6))
 ((2 4 9 5) (2 9 4 5) (9 2 4 5))
 5037
 3
 ())
```

Problème des Cavaliers XII

Conclusions :

- Le nombre de coups conduisant à une situation finale est de 26 au minimum.
- Le nombre de coups conduisant à une situation finale permutant les couleurs est de 40 au minimum.
- Toute situation est accessible en moins de 50 coups.
- Les trois situations les moins accessibles sont `(2 4 9 5)`, `(2 9 4 5)` et `(9 2 4 5)` ; 49 coups sont nécessaires.

Le problème des Cavaliers n'est que l'un des nombreux représentants de la classe des problèmes dits "de recherche" ou "d'exploration dans un espace d'état". Les problèmes de cette classe comportent tous un certain ensemble structuré (ici, les 5 040 situations possibles) et un chemin à déterminer dans cette ensemble (ici, une suite de mouvements). Fondamentalement, il n'y a aucune difficulté si ce n'est la taille de l'espace et le nombre potentiellement très élevé de chemins possibles.

Problème des Cavaliers XIII

Dans le cas présent, aucune tactique n'est requise pour réduire le travail de recherche, car 5 040 situations correspondent à un espace très réduit. Néanmoins, à titre d'illustration, nous présentons une tactique dont l'emploi est fréquemment nécessaire en pratique.

La fonction `gen` construit les chemins depuis la situation initiale jusqu'à la situation finale. Faire l'inverse ne serait ni plus ni moins efficace puisque chaque mouvement est réversible.

Ce n'est pas le cas pour tous les problèmes de recherche et, parfois, enchaîner les mouvements vers l'arrière plutôt que vers l'avant accroît très significativement l'efficacité de la recherche.

On peut attendre une amélioration de l'efficacité en combinant une recherche vers l'avant et une recherche vers l'arrière. La variante `gen2` construit les chemins à partir de leurs deux extrémités, de manière symétrique.

Problème des Cavaliers XIV

```
(define (gen2 n inits finals)
  (if (= n 0)
      (list 0 (list '() inits 0 1) (list '() finals 0 9) '() '())
      (let* ((rec1 (gen2 (- n 1) inits finals))
             (recf (cadr rec1)) (recb (caddr rec1)))
        (let ((fold (car recf)) (fnew (cadr recf))
              (flold (caddr recf)) (flnew (cadddr recf))
              (bold (car recb)) (bnew (cadr recb))
              (blold (caddr recb)) (blnew (cadddr recb)))
          (let ((fold1 (merge fold fnew))
                (fnew1 (new-succss fnew *moves* fold))
                (bold1 (merge bold bnew))
                (bnew1 (new-succss bnew *moves* bold)))
            (let ((flold1 (+ flold flnew)) (flnew1 (length fnew1))
                  (blold1 (+ blold blnew)) (blnew1 (length bnew1))
                  (i-a (inter fold1 bnew1)) (i-b (inter fnew1 bnew1)))
              (newline)
              (display (list n i-a i-b))
              (list n (list fold1 fnew1 flold1 flnew1)
                      (list bold1 bnew1 blold1 blnew1) i-a i-b)))))))
```

Problème des Cavaliers XV

La liste renvoyée comporte le nombre n = [[n]] d'itérations, un quadruplet concernant la progression vers l'avant ("f" signifie "forward"), un quadruplet analogue concernant la progression vers l'arrière ("b" signifie "backward") et deux listes [[i-a]] et [[i-b]] comportant les situations apparaissant à l'intersection des fronts avant et arrière. Le quadruplet "avant" comporte les listes des situations accessibles par l'avant en moins de n coups et en exactement n coups, et les longueurs de ces listes. La liste [[i-a]] contient les situations accessibles par l'avant en moins de n pas et, simultanément, accessibles par l'arrière en exactement n pas. De même, la liste [[i-b]] contient les situations simultanément accessibles par l'avant et par l'arrière, en exactement n pas dans les deux cas. Ces deux listes, si elles ne sont pas vides, témoignent de l'existence d'un ou plusieurs chemin(s) de longueur moindre que $2n$, ou égale à $2n$, respectivement.

Problème des Cavaliers XVI

Le nombre d'itérations passe de 49 à 25.

```
(gen2 25 (list *init*) *final*)
    (1 () ())
    ...
    (12 () ())
    (13 () ((2 9 7 8) (9 8 2 7)))
    (14 ((2 9 7 1) (9 3 2 7)) ((2 4 7 8) ... (9 7 2 3)))
    ...
    (25 ((1 6 8 0) ... (8 6 3 0)) ())


(25
 (((0 2 1 3) (0 2 3 1) ... (9 8 7 6))
  ((0 2 1 6) ... (9 8 4 0))
  2035
  191)
 (((0 1 2 3) ...  (9 8 7 6))
  ((1 6 8 0) ...  (8 6 3 0))
  5001
  24)
 ((1 6 8 0) ... (8 6 3 0))
 ())
```

Problème des Cavaliers XVII

On a vu précédemment que 26 étapes suffisaient pour passer de la situation initiale `(1 3 6 0)` à l'une des situations finales `(3 0 1 6)` et `(6 1 0 3)` ; la présente exécution montre que les chemins réalisant ce transfert ont nécessairement pour étape médiane la situation `(2 9 7 8)` ou la situation `(9 8 2 7)`.

Remarques. Dans le cas présent, aborder le problème "des deux côtés", c'est-à-dire depuis la situation initiale et depuis la situation finale, n'abrège pas la recherche, puisque tout l'espace est exploré. Notons déjà que dans beaucoup de problèmes analogues, l'espace à explorer est trop grand pour être construit en entier, ce qui peut rendre nécessaire l'emploi d'une tactique plus fine que la recherche exhaustive et systématique. Un défaut potentiel des programmes présentés ici est qu'ils ne fournissent pas la suite de mouvements permettant de passer de la situation initiale à l'une des situations finales. On peut facilement adapter les programmes pour remédier à cette lacune, mais au prix d'une consommation de ressources nettement plus élevée. La raison en est que dans une situation donnée, plusieurs mouvements sont en général possibles, ce qui donne lieu à une explosion combinatoire du nombre de chemins à construire et à explorer. Parfois, cette explosion combinatoire peut être évitée par un raisonnement simple.