Problèmes du parcours du cavalier [hs03] - Exercices

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - Problèmes du parcours du cavalier (Solution)

Mots-Clés Backtraking, Heuristique ■
Requis Axiomatique objet, Récursivité des actions, Heuristique ■
Fichiers Echiquier, Position, UtilsEchiquier, UtilsList ■
Difficulté ●●○ (4 h) ■

Objectif
Cet exercice résout les problèmes du parcours du cavalier sur un échiquier, à savoir :

- La traversée en largeur.
- La traversée récursive et itérative du cavalier d'Euler.
- L'heuristique d'accessibilité du cavalier d'Euler.
- La force brute du cavalier d'Euler.

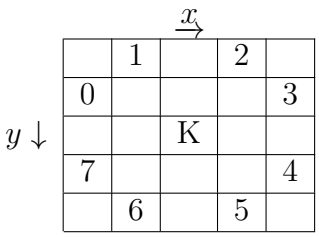
Remarque
Dans le même ordre d'idées, l'exercice @[Problème des reines sur l'échiquier] résout celui des reines.

...(énoncé page suivante)...

1 Les problèmes classiques et représentation

1.1 Les problèmes classiques

On repère chaque case d'un échiquier par ses coordonnées (x, y), celle en haut à gauche étant de coordonnées $(0, 0)$. Sur un tel échiquier, en un coup, un cavalier peut se déplacer de la case (i, j) vers celles d'entre les 8 positions qui correspondent à une case de l'échiquier (abscisse et ordonnée comprises entre 0 et 7) : $(x \pm 2, y \pm 1)$ et $(x \pm 1, y \pm 2)$.



Les problèmes classiques sont :

- **La traversée en largeur** de l'échiquier : **déterminer** si **toutes les cases** sont **accessibles à partir d'une case** (i_0, j_0) donnée, et si oui, quel est **le plus petit nombre de coups** permettant d'atteindre à partir de cette case n'importe quelle autre case de l'échiquier.
- **Le Cavalier d'Euler** dit aussi problème du « cavalier opiniâtre » : faire **passer** le cavalier **par toutes les cases** de l'échiquier **en ne visitant chaque case qu'une seule fois**.

1.2 Représentation de l'échiquier

On dispose de la classe `Echiquier` qui représente des tableaux carrés $n \times n$ à deux indices (n vaut 8 dans le cas d'un échiquier standard), ainsi que de la classe `Position` modélisant des positions (x, y).

Soit la classe `Position` modélisant des positions (x, y) de $\mathbb{Z}^2$.
C++ @[Position.hpp] @[Position.cpp]

Soit la classe `Echiquier` qui représente un tableau carré $n \times n$ d'entiers.
C++ @[Echiquier.hpp] @[Echiquier.cpp]

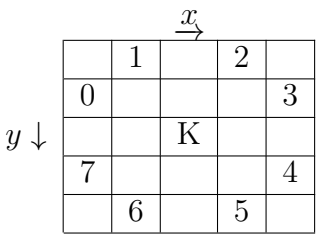
1.3 Représentation des mouvements

Les mouvements seront décrits par un tableau de `Position` selon leur composantes horizontales (axe x) et verticales (axe y). Exemple :

- Le mouvement de type 0 consiste en deux déplacements horizontaux vers la gauche et un déplacement vertical vers le haut.

- Celui de type 2 à un déplacement vers la droite et à deux déplacements verticaux vers le haut.

On représente les déplacements horizontaux vers la gauche et verticaux vers le haut par des nombres négatifs.



Si on mémorise ces « delta » de déplacements dans un `vecteur<Position>` et si la `Position pos` indique la position actuelle du cavalier, l'instruction suivante récupère dans `newpos` la nouvelle `Position` :

```
Position newpos = plusPos(pos + delta[ v ])
```

En C++, on peut aussi écrire :

```
Position newpos = pos + delta[ v ]
```

L'entier `v| dans [0..7] désigne le type de mouvement et l'opération d'addition de la classe` \lstinlinePosition@ a été surdéfini afin d'effectuer des opérations « arithmétiques » de déplacement.

Écrivez les définitions du delta de déplacements ainsi que l'opération `+| sur les` \lstinlinePosition@.

Validez vos définitions et opérations avec la solution.

Solution C++ @[pgcavalier.cpp]

```
// Tableau de deplacements: mouvements standard
const Position arrDelta0[] = {
  Position(-2,-1), Position(-1,-2),
  Position( 1,-2), Position( 2,-1),
  Position( 2, 1), Position( 1, 2),
  Position(-1, 2), Position(-2, 1)
};

// Tableau de deplacements: autres mouvements
const Position arrDelta1[] = {
  Position( 0,-3), Position( 2,-2),
  Position( 3, 0), Position( 2, 2),
  Position( 0, 3), Position(-2, 2),
  Position(-3, 0), Position(-2,-2)
};

// Vecteurs bases sur les tableaux
typedef vector<Position> CVDeplacements;
```

```
const CVDeplacements delta0(&arrDelta0[0], &arrDelta0[8]);
const CVDeplacements delta1(&arrDelta1[0], &arrDelta1[8]);

// Surdefinit l'operation + sur les Position
Position operator+(const Position& pos1, const Position& pos2)
{
  return Position(pos1.getX()+pos2.getX(), pos1.getY()+pos2.getY());
}

// Operation + sur les Position
Position plusPos(const Position& pos1, const Position& pos2)
{
  return Position(pos1.getX()+pos2.getX(), pos1.getY()+pos2.getY());
}
```

...(suite page suivante)...

2 Traversée en largeur du cavalier

Objectif
Ce problème résout le problème de la traversée en largeur du cavalier : déterminer la liste de toutes les cases accessibles à partir d'une `Position` donnée en p pas au plus.

Dessinez un échiquier de 8 par 8 sur une feuille de papier et essayez de résoudre le problème manuellement. Comment avez-vous procédé pour traiter les cellules marquées au fur et à mesure ?

Solution simple
Procédez ainsi :

- Inscrivez 1 dans la première case visitée (la case de départ).
- Puis inscrivez 2 dans toutes celles accessibles à partir de 1 qui se trouvent dans l'échiquier et non encore marquées.
- Puis 3 dans celles dans l'échiquier, accessibles à partir des 2 et non encore marquées.
- etc.

Écrivez une procédure `bfsCV(cb,delta,depart)` qui parcourt en largeur un `Echiquier cb` à partir d'une `Position depart` selon les `CVDeplacements delta`.

Aide simple
Utilisez une file mémorisant les `Position` à examiner.

Outil C++
Le modèle de classes `queue<T>` est défini dans la bibliothèque `<queue>`.

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
   Parcourt en largeur d'un Echiquier a partir d'une Position
   @param[in,out] cb - un Echiquier
   @param[in] delta - delta de deplacements
   @param[in] depart - Position de depart
*/
void bfsCV(Echiquier& cb, const CVDeplacements& delta, const Position& depart)
{
  // Reinitialise l'echiquier
  cb.reset(0);

  // Se positionne au point de depart et fixe sa valeur (marque)
  cb.set(depart, 1);
```

```
  // Definit la file des positions a examiner et enfile la
  // position de depart
  queue<Position> q;
  q.push(depart);

  // Parcourt tantque possible
  while( not q.empty() )
  {
    // Recupere la "prochaine" position (x,y) de la file
    // pour la traiter puis la defile
    Position pos = q.front();
    q.pop();

    // Recupere sa valeur sur l'echiquier
    int val = cb.get(pos);

    // Marque toutes les cases accessibles a partir d'elles
    for (unsigned nv = 0; nv < delta.size(); ++nv)
    {
      Position nextpos = pos + delta[nv];
      // Si nextpos est dans le domaine et non encore marquee,
      // alors visite la cellule et l'enfile dans la queue
      if (cb.contient(nextpos) && cb.get(nextpos) == 0)
      {
        cb.set(nextpos, val+1);
        q.push(nextpos);
      }
    }
  }
}
```

Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsList.cpp]

Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

```
#include "UtilsList.cpp"
using namespace UtilsList;
```

Soit la procédure générique `afficherList(ls)` qui affiche une `list<T> ls`.

C++ @[afficherList] (dans UtilsList.cpp)

Écrivez une procédure `afficherListeBFS(cb)` qui affiche la liste des cellules accessible d'un `Echiquier cb` et qui calcule et affiche la profondeur de la visite.

Outil C++

Le modèle de classes `list<T>` est définie dans la bibliothèque `<list>`.

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
  Affiche la liste des cellules accessibles et calcule
  la profondeur de la visite
  @param[in] cb - un Echiquier
*/
void afficherListeBFS(const Echiquier& cb)
{
  list<Position> cells;
  int ncoups = 0;
  for (unsigned iy = 0; iy < cb.sizeN(); ++iy)
  {
    for (unsigned ix = 0; ix < cb.sizeN(); ++ix)
    {
      Position c(ix, iy);
      if ( cb.get(ix, iy) != 0 )
      {
        cells.push_back(c);
        ncoups = max(ncoups, cb.get(c));
      }
    }
  }
  cout<<cells.size()<<" positions accessibles en "
      <<ncoups<<" coups:"<<endl;
  afficherList(cells);
}
```

Téléchargez le fichier suivant et mettez-le dans votre dossier.
C++ @[UtilsEchiquier.cpp]

Copiez/collez ensuite les lignes suivantes :
C++ **Au début** de votre programme :

```
#include "UtilsEchiquier.cpp"
using namespace UtilsEchiquier;
```

Soit la procédure `afficherEchiquier(cb)` qui affiche un `Echiquier cb`.
C++ @[afficherEchiquier] (dans UtilsEchiquier.cpp)

Écrivez une procédure `test_bfsCV` qui :

- Demande et saisit la dimension de l'échiquier dans un entier `n`.
- Instancie un `Echiquier` d'ordre `n`.
- Demande et saisit la position de départ.
- Lance la traversée en largeur de l'échiquier à partir de cette position de départ.
- Enfin affiche l'échiquier puis la liste des cellules accessibles.

Testez. Exemple d'exécution :

```
Dimension n de l'echiquier? 10
Mouvements standard (0 == oui)? 0
Position (x,y) de depart? (0,0=
Traversee en largeur:
  1  4  3  4  3  4  5  6  5  6
  4  5  2  3  4  5  4  5  6  7
  3  2  5  4  3  4  5  6  5  6
  4  3  4  3  4  5  4  5  6  7
  3  4  3  4  5  4  5  6  5  6
  4  5  4  5  4  5  6  5  6  7
  5  4  5  4  5  6  5  6  7  6
  6  5  6  5  6  5  6  7  6  7
  5  6  5  6  5  6  7  6  7  8
  6  7  6  7  6  7  6  7  8  7

100 positions accessibles en 8 coups:
((0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0)
 (8,0) (9,0) (0,1) (1,1) (2,1) (3,1) (4,1) (5,1)
 (6,1) (7,1) (8,1) (9,1) (0,2) (1,2) (2,2) (3,2)
 (4,2) (5,2) (6,2) (7,2) (8,2) (9,2) (0,3) (1,3)
 (2,3) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3) (9,3)
 (0,4) (1,4) (2,4) (3,4) (4,4) (5,4) (6,4) (7,4)
 (8,4) (9,4) (0,5) (1,5) (2,5) (3,5) (4,5) (5,5)
 (6,5) (7,5) (8,5) (9,5) (0,6) (1,6) (2,6) (3,6)
 (4,6) (5,6) (6,6) (7,6) (8,6) (9,6) (0,7) (1,7)
 (2,7) (3,7) (4,7) (5,7) (6,7) (7,7) (8,7) (9,7)
 (0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8)
 (8,8) (9,8) (0,9) (1,9) (2,9) (3,9) (4,9) (5,9)
 (6,9) (7,9) (8,9) (9,9) )
```

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
void test_BFS()
{
  // Saisit des donnees
  cout<<"Dimension n de l'echiquier? ";
  unsigned n;
  cin >> n;
  Echiquier cb(n);
  cout<<"Mouvements standard (0 == oui)? ";
  unsigned rep;
  cin >> rep;
  cout<<"Position (x,y) de depart? ";
  Position cell;
  cin >> cell;
  // Lance la traversee en largeur
```

```
  if (rep == 0)
  {
    bfsCV(cb, delta0, cell);
  }
  else
  {
    bfsCV(cb, delta1, cell);
  }
  // Affiche le resultat
  cout<<"Traversee en largeur: "<<endl;
  afficherEchiquier(cb, 3);
  afficherListeBFS(cb);
}
```

Estimez la complexité de votre procédure de traversée.

Solution simple

C'est en $O(n^2)$ où n est l'ordre du damier.

...(suite page suivante)...

3 Problème du cavalier d'Euler

Objectif
Proposé pour la première fois par le mathématicien Euler, le tour du cavalier Euler représente l'un des casse-têtes les plus intéressants pour les mordus des échecs. Il s'agit de répondre à la question : « Est-il possible de déplacer le cavalier sur l'intégralité des 64 cases (cas d'un échiquier standard) d'un jeu d'échec vide, sans jamais revenir sur une même case ? »

Ce problème détermine **tous** les échiquiers-solutions.

3.1 Résolution récursive

Stratégie récursive
L'algorithme de parcours en utilisant une stratégie récursive est :

- Initialement le cavalier se trouve sur une case de l'échiquier.
- Partant de cette case, théoriquement huit mouvements sont possibles et ceux-ci sont examinés successivement, à moins qu'entre temps la solution du problème n'ait été trouvée.

Dessinez un échiquier de 5 par 5 sur une feuille de papier et essayez de résoudre le problème manuellement.

Solution simple
Inscrivez 1 dans la première case visitée (la case de départ), 2 dans la deuxième, 3 dans la troisième, etc.

Écrivez une procédure `dfsCV(cb,delta,depart,sols)` qui parcourt en profondeur un `Echiquier cb` à partir d'une `Position depart` selon les `CVDeplacements delta` et qui mémorise tous les échiquiers-solutions dans une `Liste<Echiquier> sols`.

Outil C++
Le modèle de classes `list<T>` est défini dans la bibliothèque `<list>`.

Aide simple
Écrivez une procédure récursive `dfsCVRec(cb,delta,pos,p,sols)` qui tente la traversée d'un `Echiquier cb` à partir d'une `Position pos`, numéro de pas `p`.

Aide détaillée
Plus précisément :

- On suppose qu'à l'appel de la procédure, la `Position` est dans l'échiquier et non encore marquée : on marque la cellule.

- Est-ce une configuration solution : si oui on l'enfile dans la liste des échiquiers-solutions.
- S'il faut continuer la recherche, on tente la traversée en chacune des positions possibles. On calcule les coordonnées de la case destination suivante éventuelle. Si cette case est valide, c.à-d. si elle se trouve bien dans les limites de l'échiquier et qu'elle est libre, cette case peut être occupée par le cavalier : on fait un appel récursif de la procédure à partir de cette `Position` en la marquant avec le numéro de pas suivant.
- Si partant d'une case, il n'est pas possible de déplacer le cavalier au coup suivant, cet appel se termine après avoir épuisé les huit possibilités de mouvement du cavalier : la case sur laquelle il se trouve actuellement doit être libérée (en la marquant avec sa valeur initiale 0) puisqu'il lui est impossible de continuer partant de cette case. Le cavalier retourne donc à la case d'où il est venu, puis il envisage le mouvement suivant parmi les huit possibles.

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
  Traversee en profondeur d'un Echiquier
  @param[in,out] cb - un Echiquier
  @param[in] delta - delta de deplacements
  @param[in] pos - la Position du cavalier
  @param[in] p - numero du pas
  @param[in,out] sols - la liste des solutions
  @pre pos est dans l'echiquier et non encore marque
*/
void dfsRecCV(Echiquier& cb, const CVDeplacements& delta,
  const Position& pos, unsigned p, list<Echiquier>& sols)
{
  // Marque la cellule
  cb.set(pos, p);

  // Est-ce une solution?
  if (cb.sizeN()*cb.sizeN() == p)
  {
    sols.push_back(cb);
  }

  // Sinon tente la traversee en chacune des positions possibles
  for (unsigned nv = 0; nv < delta.size(); ++nv)
  {
    Position nextpos = pos + delta[nv];
    if (cb.contient(nextpos) && cb.get(nextpos) == 0)
    {
      dfsRecCV(cb, delta, nextpos, p+1, sols);
    }
  }

  // Libere la cellule
  cb.set(pos, 0);
}
```

```
/**
  Parcours en profondeur, version recursive
  Lance la traversee a partir d'une position depart
  @param[in,out] cb - un Echiquier
  @param[in] delta - delta de deplacements
  @param[in] depart - Position de depart
  @param[out] sols - la liste des solutions
*/
void dfsCV(Echiquier& cb, const CVDeplacements& delta,
    const Position& depart, list<Echiquier>& sols)
{
  // Initialise l'echiquier
  cb.reset(0);

  // Lance la traversee
  sols.clear();
  if (cb.contient(depart))
  {
    dfsRecCV(cb, delta, depart, 1, sols);
  }
}
```

Écrivez une procédure `afficherListeDFS(sols)` qui affiche la liste des échiquiers-solutions `sols` issue de la traversée du cavalier d'Euler.

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
  Affiche les solutions de la traversee
  @param[in] sols - la liste des Echiquier-solutions
*/
void afficherListeDFS(const list<Echiquier>& sols)
{
  cout<<"Traversee en profondeur: nombre de solutions = "<<sols.size()<<endl;
  int nsols;
  do{
    cout<<"Nombre de solutions a afficher? ";
    cin >> nsols;
  } while (nsols < 0 || sols.size()< nsols);
  list<Echiquier>::const_iterator it = sols.begin();
  for (int k = 1; k <= nsols; ++k, ++it)
  {
    afficherEchiquier(*it, 3);
  }
}
```

Copiez/collez la procédure `test_bfsCV` en la procédure `test_dfsCV` puis modifiez-la de sorte à réaliser la traversée en profondeur.

Testez. Exemple d'exécution :

```
Dimension n de l'echiquier? 5
Mouvements standard (0 == oui)? 0
Position (x,y) de depart? (0,0)
Traversee en profondeur: nombre de solutions = 304
Nombre de solutions a afficher? 2
 1 16 11  6  3
10  5  2 17 12
15 22 19  4  7
20  9 24 13 18
23 14 21  8 25

 1 16 11  6  3
10  5  2 21 12
15 22 17  4  7
18  9 24 13 20
23 14 19  8 25
```

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
void test_DFS()
{
  // Saisit des donnees
  cout<<"Dimension n de l'echiquier? ";
  unsigned n;
  cin >> n;
  Echiquier cb(n);
  cout<<"Mouvements standard (0 == oui)? ";
  unsigned rep;
  cin >> rep;
  cout<<"Position (x,y) de depart? ";
  Position cell;
  cin >> cell;
  // Lance la traversee en profondeur
  list<Echiquier> sols;
  if (rep == 0)
  {
    dfsCV(cb, delta0, cell, sols);
  }
  else
  {
    dfsCV(cb, delta1, cell, sols);
  }
  // Affiche le resultat
  afficherListeDFS(sols);
}
```

3.2 Résolution itérative

L'algorithme de parcours en utilisant une stratégie itérative dérive de la version récursive.

Écrivez une procédure **itérative** `dfsIterCV(cb,delta,depart,sols)` qui parcourt en profondeur un `Echiquier cb` à partir d'une `Position depart` selon les `CVDeplacements delta`. La procédure mémorise tous les échiquiers-solutions dans une `Liste<Echiquier> sols`.

Aide simple

Utilisez une pile mémorisant un couple `(Position,direction)`.

Outil C++

Le modèle de classes `stack<T>` est défini dans la bibliothèque `<stack>`.

Aide détaillée

Procédez comme suit :

- Définissez une pile stockant des `std::pair<Position,int>` l'entier indiquant le numéro de la direction testée (voir la boucle `Pour` dans la version récursive).
- Démarrez la traversée à partir de la `Position` en la marquant avec le numéro de pas 1, puis l'empiler (avec l'indice de direction -1).
- Tant que la traversée est possible (c.-à-d. pile non vide et nombre maximum de configurations solutions non atteint) faire :
  - Récupérez les informations du sommet de la pile pour continuer à traiter la `Position` à partir de la prochaine direction.
  - S'il existe une cellule disponible à partir de la position courante, continuez la traversée à partir de cette nouvelle `Position`.
  - Sinon, en cas d'échec, c'est un cul de sac : dépilez le couple et libérez la cellule sur l'échiquier (en la marquant avec sa valeur initiale 0).

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
  Parcours en profondeur, version iterative
  @param[in,out] cb - un Echiquier
  @param[in] delta - delta de deplacements
  @param[in] depart - Position de depart
  @param[out] sols - la liste des Echiquier-solutions
*/
void dfsIterCV(Echiquier& cb, const CVDeplacements& delta,
    const Position& depart, list<Echiquier>& sols)
{
  // Reinitialise l'echiquier
  cb.reset(0);
```

```
  // Couplet sur la pile: (position, # direction)
  typedef pair<Position, unsigned> Couplet;
  stack<Couplet, vector<Couplet> > pile;

  // Demarre la traversee a partir de depart
  if (cb.contient(depart))
  {
    cb.set(depart, 1);
    pile.push( make_pair(depart, -1) );
  }

  // Traverse tantque possible
  while( not pile.empty() ) // && sols.size() < mxsols)
  {
    // Recupere les informations du sommet de la pile
    // pour continuer a traiter la position pos a partir
    // de la prochaine direction
    Position pos = pile.top().first;
    unsigned ndir = pile.top().second;

    // Cherche la prochaine position disponible issue
    // de pos selon la direction ndir suivante
    Position nextpos;
    while (++ndir < delta.size())
    {
      nextpos = pos + delta[ndir];
      if (cb.contient(nextpos) && cb.get(nextpos) == 0)
      {
        break;
      }
    }

    // En cas de recherche frutueuse, continue la traversee a
    // partir de la position nextpos. Sinon cas d'echec: libere
    // la position pos et revient en arriere (backtracking)
    if (ndir < delta.size())
    {
      // Notifie la direction
      pile.top().second = ndir;

      // Valeur de la cellule
      int n = cb.get(pos) + 1;

      // Marque la cellule
      cb.set(nextpos, n);

      // Est-ce une solution? Si oui
      if (n >= cb.sizeN()*cb.sizeN())
      {
        // Enfile la solution
        sols.push_back(cb);

//        // Continue la recherche?
//        if (sols.size() >= mxsols)
//        {
//          break;
//        }
```

```
      }

      // Empile le couplet suivant
      pile.push(make_pair(nextpos, -1));
    }
    else
    {
      // Ici: cul de sac: depile le couplet
      pile.pop();

      // Libere la position
      cb.set(pos, 0);
    }
  }
  // Fin de traversee: si la pile est vide, la traversee est
  // un parcours exhaustif: le systeme detruit la pile:
}
```

Copiez/collez la procédure `test_dfsCV` en la procédure `test_dfsIterCV` puis modifiez-la de sorte à appeler la procédure itérative de la traversée en profondeur.

Testez. Exemple d'exécution :

```
Dimension n de l'echiquier? 5
Mouvements standard (0 == oui)? 0
Position (x,y) de depart? (0,0)
Traversee en profondeur: nombre de solutions = 304
Nombre de solutions a afficher? 2
  1 16 11  6  3
 10  5  2 17 12
 15 22 19  4  7
 20  9 24 13 18
 23 14 21  8 25

  1 16 11  6  3
 10  5  2 21 12
 15 22 17  4  7
 18  9 24 13 20
 23 14 19  8 25
```

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
void test_DFSIter()
{
  // Saisit des donnees
  cout<<"Dimension n de l'echiquier? ";
  unsigned n;
  cin >> n;
```

```
  Echiquier cb(n);
  cout<<"Mouvements standard (0 == oui)? ";
  unsigned rep;
  cin >> rep;
  cout<<"Position (x,y) de depart? ";
  Position cell;
  cin >> cell;
  // Lance la traversee en profondeur
  list<Echiquier> sols;
  if (rep == 0)
  {
    dfsIterCV(cb, delta0, cell, sols);
  }
  else
  {
    dfsIterCV(cb, delta1, cell, sols);
  }
  // Affiche le resultat
  afficherListeDFS(sols);
}
```

...(suite page suivante)...

4 Heuristique du cavalier d'Euler

Objectif
Ce problème présente une stratégie de déplacement du cavalier à l'aide d'une approche **heuristique** (ou stratégique). Cette nouvelle approche ne garantit pas le succès mais, si elle est bien choisie, elle augmente de manière significative nos chances de réussite.

4.1 Heuristique de déplacement

Heuristique d'accessibilité
Vous avez sûrement remarqué la difficulté d'atteindre les cases situées sur le bord de l'échiquier. Les cases les plus gênantes et inaccessibles se situent aux quatre coins de ce dernier. L'intuition peut suggérer d'atteindre les cases gênantes en premier et de conserver les plus accessibles en dernier, lorsque l'échiquier est presque entièrement visité.

Ce problème développe l'**heuristique d'accessibilité** en classifiant les cases selon la facilité avec laquelle le cavalier peut les visiter. Lors du choix des mouvements du cavalier, on privilégie les cases les plus inaccessibles. On calcule un `Echiquier` du nombre de cases à partir desquelles chaque case est accessible.

Exemple
Dans le cas d'un échiquier d'ordre $n \geq 5$, les accessibilités varient de 8 au centre de l'échiquier vide, à 2 aux quatre coins de ce dernier et à 3, 4 ou 6 pour les autres cases comme suit :

$n = 5$

2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2

$n = 6$

a	b	c			
		d			
		e			

$n = 8$

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Calculez les accessibilités du tableau pour $n = 6$ des six points spécifiés.

Écrivez une procédure `initHeuristiq(hb,delta)` qui calcule le tableau des accessibilités `Echiquier hb` selon les `CVDeplacements delta`.

Écrivez une procédure `heuristiqCV(cb,delta,depart,hb)` qui traverse un `Echiquier cb` à partir d'une `Position depart` selon les `CVDeplacements delta` et qui exploite l'heuristique d'accessibilité en modifiant les accessibilités `Echiquier hb`.

Aide simple

Le cavalier doit se déplacer vers la case qui possède l'accessibilité la plus faible. En cas d'égalité, le cavalier choisit n'importe quel mouvement. Diminuez l'accessibilité de chacune des cases au fur et à mesure que le cavalier se déplace. De cette façon, à tout moment du parcours, le nombre d'accessibilité de chaque case encore accessible reste égal au nombre de cases qui permettent d'accéder à celle-ci, les maintenant à des valeurs correctes.

Validez vos procédures avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
  Reinitialise l'echiquier et le tableau des accessibilites
  @param[out] cb - un Echiquier
  @param[out] hb - Echiquier des accessibilites
  @param[in] delta - delta de deplacements
*/
void initHeuristiq(Echiquier& cb, Echiquier& hb, const CVDeplacements& delta)
{
  // Reinitialise l'echiquier de parcours du cavalier
  cb.reset(0);

  // Reinitialise l'accessibilite de chacune des positions
  hb.reset(0);
  for (unsigned y = 0; y < cb.sizeN(); ++y)
  {
    for (unsigned x = 0; x < cb.sizeN(); ++x)
    {
      Position pos(x, y);
      unsigned a = 0;
      for (unsigned nv = 0; nv < delta.size(); ++nv)
      {
        Position nextpos = pos + delta[nv];
        if ( cb.contient(nextpos) )
        {
          ++a;
        }
      }
      hb.set(pos, a);
    }
  }
}
```

```
/**
  Lance la traversee heuristique
  @param[in,out] cb - un Echiquier
  @param[in] delta - delta de deplacements
  @param[in] depart - Position de depart
  @param[in,out] hb - Echiquier des accessibilites
*/
void heuristiqCV(Echiquier& cb, const CVDeplacements& delta,
    const Position& depart, Echiquier& hb)
{
  // Reinitialise les donnees
```

```
  initHeuristiq(cb, hb, delta);

  // Demarre la traversee a partir de depart
  if ( !cb.contient(depart) )
  {
    return;
  }

  // Traverse tantque possible
  const unsigned NULL_ACCESS = delta.size()+1;
  // position courante
  Position pos = depart;
  // pas courant
  int step = 0;
  while(true)
  {
    // Marque la position
    cb.set(pos, ++step);
    hb.set(pos, NULL_ACCESS);

    // Recherche la prochaine position
    Position nextpos;
    // Et sa valeur d'accessibilite
    unsigned nacces = NULL_ACCESS;
    for (unsigned nv = 0; nv < delta.size(); ++nv)
    {
      // Calcule la position destination
      Position nvpos = pos + delta[nv];

      // Si dans l'echiquier et non marquee
      if ( cb.contient(nvpos) && cb.get(nvpos) == 0 )
      {
        // Decremente l'accessibilite de nvpos puis
        // teste si c'est la suivante
        unsigned val = hb.get(nvpos);
        hb.set(nvpos, --val);
        if (val < nacces)
        {
          nacces = val;
          nextpos = nvpos;
        }
      }
    }

    // En cas d'echec: fin de la traversee
    if (NULL_ACCESS == nacces)
    {
      break;
    }

    // Sinon, on se deplace sur la prochaine position
    pos = nextpos;
  }
}
```

4.2 Procédure de test

Lorsque le cavalier peut atteindre la case de départ à partir de la dernière case, il effectue le « tour complet » du cavalier.

Écrivez une procédure `afficherHeuristiq(cb,hb)` qui affiche le résultat d'une traversée heuristique d'un `Echiquier cb` et d'accessibilités `Echiquier hb`. Dans le cas où le tour complet n'est pas réalisé, affichez la valeur du dernier pas.

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```
/**
  Affiche le resultat
  @param[in] cb - un Echiquier
  @param[in] hb - Echiquier des accessibilites
*/
void afficherHeuristiq(const Echiquier& cb, const Echiquier& hb)
{
  // Recherche du dernier pas
  int step = 0;
  for (unsigned y = 0; y < cb.sizeN(); ++y)
  {
    for (unsigned x = 0; x < cb.sizeN(); ++x)
    {
      if (cb.get(x, y) > step)
      {
        step = cb.get(x, y);
      }
    }
  }
  if (step == cb.sizeN() * cb.sizeN())
  {
    cout<<"SOLUTION trouvee"<<endl;
  }
  else
  {
    cout<<"Dernier pas: "<<step<<endl;
  }
  cout<<"Echiquier final: "<<endl;
  afficherEchiquier(cb, 3);
  cout<<" Dernier HB: "<<endl;
  afficherEchiquier(hb, 3);
}
```

Copiez/collez la procédure `test_dfsCV` en la procédure `test_heuristiqCV` puis modifiez-la de sorte à réaliser la traversée heuristique.

Testez. Exemple d'exécution :

```
Dimension n de l'echiquier? 5
Mouvements standard (0 == oui)? 0
Position (x,y) de depart? (0,0)
SOLUTION trouvee
Echiquier final:
  1 14  9 20  3
 24 19  2 15 10
 13  8 25  4 21
 18 23  6 11 16
  7 12 17 22  5

 Dernier HB:
  9  9  9  9  9
  9  9  9  9  9
  9  9  9  9  9
  9  9  9  9  9
  9  9  9  9  9
```

Validez votre procédure avec la solution.

Solution C++ @[pgcavalier.cpp]

```cpp
void test_Heuristiq()
{
  // Saisit des donnees
  cout<<"Dimension n de l'echiquier? ";
  unsigned n;
  cin >> n;
  Echiquier cb(n);
  cout<<"Mouvements standard (0 == oui)? ";
  unsigned rep;
  cin >> rep;
  cout<<"Position (x,y) de depart? ";
  Position cell;
  cin >> cell;
  // Lance la traversee heuristique
  Echiquier hb(n);
  if (rep == 0)
  {
    heuristiqCV(cb, delta0, cell, hb);
  }
  else
  {
    heuristiqCV(cb, delta1, cell, hb);
  }
  // Affiche le resultat
  afficherHeuristiq(cb, hb);
}
```

4.3 Amélioration de la traversée heuristique

Testez en commençant la partie à l'un des quatre coins.
Avez-vous visité toutes les cases ?

Modifiez le point de départ du cavalier pour qu'il commence successivement à partir des $n \times n$ cases de l'échiquier. Combien de fois avez-vous résolu le problème en visitant toutes les cases ?

Écrivez une nouvelle version `heuristiqCVSeq(cb,delta,depart,hb)` qui, en cas d'égalité d'accessibilité pour les mouvements actuels, regarde les accessibilités des mouvements suivants et qui choisit la séquence des deux mouvements qui atteint la plus faible accessibilité.

...(suite page suivante)...

5 Force brute du cavalier d'Euler

Objectif

Dans le @[Problème du cavalier d'Euler] et @[Heuristique du cavalier d'Euler], nous avons développé une solution au problème du tour du cavalier. L'approche suggérée de l'heuristique d'accessibilité génère plusieurs solutions et s'exécute avec efficacité. La stratégie gloutonne exhaustive @[Traversée en largeur du cavalier] permet de générer toutes les solutions.

Comme la puissance des ordinateurs ne cesse de croître, on peut résoudre ce problème avec des algorithmes relativement peu sophistiqués et exploiter la puissance de calcul de la machine. On appelle cette résolution de problème : l'approche « par force brute ».

A l'aide de la génération de nombres aléatoires, écrivez une procédure `forceBruteCV(cb,delta,depart)` qui déplace le cavalier sur un `Echiquier cb` à partir d'une `Position depart` selon les `CVDeplacements delta`. La procédure doit tirer au hasard la case suivante tout en respectant les déplacements possibles.

Copiez/collez la procédure `test_dfsCV` en la procédure `test_forceBruteCV` puis modifiez-la de sorte à réaliser la traversée par force brute. La procédure essaie un tour et affiche ses visites dans l'échiquier.

Testez. Votre cavalier a-t-il réussi le tour complet ?

La procédure précédente n'a probablement pas parcouru un grand nombre de cases. Modifiez-la pour qu'elle essaie x (par exemple 1000) tours. Utilisez un tableau à un seul indice pour conserver le nombre d'essais de chaque tour. Affichez les résultats sous format tabulaire lorsque les x tours sont complétés. Quel est le meilleur résultat ?

La procédure précédente a vraisemblablement parcouru un grand nombre de cases, sans jamais compléter un tour complet. Enlevez les conditions d'arrêt pour qu'elle tente un nouveau tour tant qu'elle n'a pas effectué un tour complet. Cette nouvelle version peut s'exécuter pendant des heures, même sur un ordinateur puissant. Conservez et affichez le nombre d'essais de chaque tour lorsque le premier tour complet est trouvé. Combien de tours votre procédure a-t-elle essayé avant d'obtenir une solution complète ? Pendant combien de temps votre procédure s'est-elle exécuté ?

Comparez la version de l'approche par force brute avec l'approche de l'heuristique d'accessibilité. Laquelle demande une étude plus approfondie du problème ? Quel algorithme entraîne le plus de problèmes à développer ? Lequel requiert le plus de puissance de calculs ? Sommes-nous assurés d'obtenir une solution avec la force brute ? Avec l'heuristique d'accessibilité ? Discutez des forces et des faiblesses de l'approche par force brute en général.

6 Références générales

Comprend [Bajard-AL1, parcours de tableaux] ■