Jouer à tirs croisés [je13] - Exercice

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Université Haute-Alsace

Version 22 mai 2018

Table des matières

C++ - Jouer à tirs croisés (Solution)

Mots-Clés Jeu avec stratégie ■
Requis Axiomatique impérative (sauf Fichiers) ■
Difficulté ●●○ (6 h) ■

Objectif
Cet exercice réalise le jeu dénommé *Tirs Croisés*, paru dans [Jeux et Stratégies :n7].

1 Présentation du jeu

Le jeu « Tirs Croisés »

C'est un jeu tactique qui se joue sur un échiquier comprenant $n \times n$ cases (par exemple 5×5). On prépare le jeu en disposant des entiers pseudo-aléatoires compris entre 1 et 9 sur la grille. Pratiquement, on peut matérialiser les entiers par des cartes à jouer de valeurs correspondantes.

A tour de rôle, chaque joueur prend une carte. Le premier joueur choisit celle qu'il désire n'importe où, mais le second doit nécessairement prélever une carte dans la ligne ou dans la colonne où se trouvait la carte prise par son adversaire. Dans la version programmée, c'est la machine qui tire au hasard la position de départ et le joueur qui commence.

Le jeu se poursuit en respectant cette règle jusqu'à épuisement des cartes, ou jusqu'à ce que l'on ne puisse plus prendre (plus de carte ni dans la ligne, ni dans la colonne). Le vainqueur est alors celui dont le total des points prélevés est maximum.

Exemple

Voici le début d'une partie de Tirs-Croisés. Une position `(x,y)` désigne la case (colonne `x`, ligne `y`). Le curseur est identifié par un `#`.

- La machine propose le tableau suivant. Le curseur est en (2,1) et c'est à l'humain de commencer.

y↓ x→	1	2	3	4	5
1	7	#	4	5	6
2	1	5	8	5	7
3	4	5	6	9	1
4	4	3	2	6	2
5	9	9	8	7	3

- L'humain doit jouer dans la colonne 2 ou ligne 1. Il prend le 9 en (2,5). Le score actuel de l'humain : 9.

y↓ x→	1	2	3	4	5
1	7	.	4	5	6
2	1	5	8	5	7
3	4	5	6	9	1
4	4	3	2	6	2
5	9	#	8	7	3

- La machine doit jouer dans la colonne 2 ou ligne 5. Elle prend le 9 en (1,5). Le score actuel de la machine : 9. Le tableau est alors :

y↓ x→	1	2	3	4	5
1	7	.	4	5	6
2	1	5	8	5	7
3	4	5	6	9	1
4	4	3	2	6	2
5	#	.	8	7	3

- L'humain prend le 7 en (1,1). Le score actualisé est : 9+7=16.

y↓ x→	1	2	3	4	5
1	#	.	4	5	6
2	1	5	8	5	7
3	4	5	6	9	1
4	4	3	2	6	2
5	.	.	8	7	3

- La machine prend le 6 en (5,1). Son score actualisé est : 9+6=15. Le tableau actualisé est :

y↓ x→	1	2	3	4	5
1	.	.	4	5	#
2	1	5	8	5	7
3	4	5	6	9	1
4	4	3	2	6	2
5	.	.	8	7	3

...(suite page suivante)...

2 Modélisation et Découpage

2.1 Modélisation de l'échiquier

Analyse

La grille de jeu présente plusieurs types de données. D'une part, il s'agit des objets dont la valeur est réalisée de manière aléatoire en début de jeu. Et d'autre part, il y a le curseur déplacé par les joueurs, chacun à tour de rôle. Les cases ne contenant aucune information doivent également être modélisées. A chacune de ces informations, nous pouvons associer un entier, ce qui permet de coder de manière homogène les informations :

- Les objets sont codés par leur valeur – un entier strictement positif leur est associé.
- Le curseur est codé par un entier négatif afin de le distinguer des objets.
- Les cases vides sont codées par l'entier 0.

Toutes les cases doivent être modélisées afin de représenter la grille et ceci sous la forme d'un tableau bidimensionnel, comportant huit lignes et huit colonnes. La grille peut être dimensionnée différemment, ce qui amène à utiliser des constantes pour les tailles des deux dimensions. De manière générale, si l'application prévoit une configuration de la grille de jeu, alors ces valeurs seront données par des variables.

Le joueur sélectionne une case de la grille sur la colonne ou la ligne du curseur. Celle-ci est identifiée par son numéro de ligne et son numéro de colonne, les deux supérieurs à 1. Ces numéros seront mis en corrélation avec les indices des lignes et des colonnes de la matrice utilisée pour la modélisation.

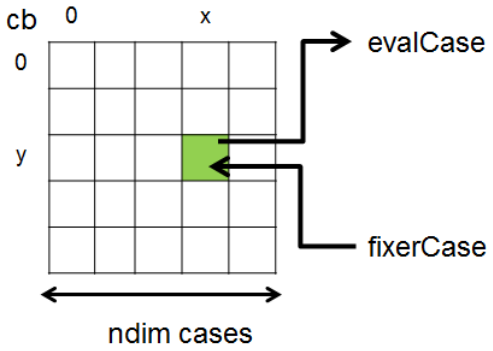
Définissez les valeurs constantes de la case `VIDE=0`, la case `CURSEUR=-1` et la case du `BORD=-2`. Rappel : les entiers 1, 2, etc. identifient la valeur des objets.

Type Echiquier

Le plateau (tableau bidimensionnel) sera représenté par un type.

(0.5 point) Définissez les constantes `TMIN=4` (taille minimale) et `TMAX=8` (taille maximale) du plateau d'un échiquier puis le type `Echiquier` modélisant le plateau dont la dimension effective sera un entier compris entre `TMIN` et `TMAX` (inclus).

(1 point) L'accès à une case en `(x,y)` (colonne,ligne) d'un `Echiquier` se fera via une opération `evalCase(...)` et la modification par une opération `fixerCase(...)`. Écrivez ces deux opérations puis justifiez leur intérêt.



Solution simple

Les opérations permettent de devenir indépendant de la modélisation de l'`Echiquier`. En effet, on peut modéliser un tableau bidimensionnel sous une forme linéaire.

Validez vos définitions et opérations avec la solution.

Solution C++ @[UtilsEchiquier.cpp]

```
/** Type Echiquier */
const int TMIN = 4;  //! taille minimale du plateau
const int TMAX = 20; //! taille maximale du plateau
struct Echiquier
{
  int cases[TMAX+1][TMAX+1]; //! le plateau
  int ndim; //! dimension effective
};
```

```
/**
  Valeur d'une case [coords valides] d'un echiquier
  @param[in] cb - echiquier dont on lit la case
  @param[in] x - colonne de la case (valide)
  @param[in] y - ligne de la case (valide)
  @return la valeur de la case de coordonnees (x,y) de cb
*/
int evalCase(const Echiquier& cb, int x, int y)
{
  return (cb.cases[y][x]);
}
```

```
/**
  Fixe la valeur d'une case [coords valides] d'un echiquier
  @param[in,out] cb - Echiquier dans lequel on fixe la valeur
  @param[in] x - colonne de la case (valide)
  @param[in] y - ligne de la case (valide)
  @param[in] valeur - valeur a fixer
*/
void fixerCase(Echiquier& cb, int x, int y, int valeur)
{
  cb.cases[y][x] = valeur;
}
```

(0.25 point) Définissez les constantes de la case `VIDE=???` (<- à vous de voir), la case `DETRUITE=???` et la case du `BORD=???`. Rappel : les entiers 1, 2, etc. identifient les (pions des) joueurs.

Validez vos définitions avec la solution.

Solution C++ @[UtilsEchiquier.cpp]

```
/** Constantes identifiant les cases speciales */
const int VIDE = 0; //! case VIDE
const int DETRUITE = -1; //! Jeu Isola
const int CURSEUR = -1; //! Jeu Tirs
const int BORD = -2; //! case BORD
```

(0.25 point) Écrivez le **profil** d' une opération `initialiserEchiquier(...)` qui initialise un `Echiquier`. Le tableau ne doit contenir que des cases `VIDE`s et des cases de `BORD`.

Validez votre opération avec la solution.

Solution C++ @[UtilsEchiquier.cpp]

```
/**
  Initialise un Echiquier
  @param[out] cb - un Echiquier
  @param[in] n - taille du plateau de l'echiquier
*/
void initialiserEchiquier(Echiquier& cb, int n)
{
  cb.ndim = n;
  for (int y = 1; y <= cb.ndim; ++y)
  {
    for (int x = 1; x <= cb.ndim; ++x)
    {
      fixerCase(cb, x, y, VIDE);
    }
  }
  for (int k = 0; k <= cb.ndim; ++k)
  {
    fixerCase(cb, 0, k, BORD); // ligne du haut
    fixerCase(cb, k, 0, BORD); // colonne de gauche
  }
}
```

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

C++ @[UtilsConsole.cpp]

Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

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

Soient les procédures d'entrée/sortie en mode texte de l'écran :

- Procédure `clrscr()` : Efface l'écran
- Procédure `gotoxy(x,y)` : Positionne le curseur en (colonne `x`, ligne `y`)

C++ @[UtilsConsole.cpp]

Définissez les constantes `OFFSET_X=2` et `OFFSET_Y=2`, offsets X et Y d'affichage d'une case de l'échiquier, de sorte qu'une case sera toujours affiché au même endroit.

(0.25 point) Écrivez le **profil** d' une opération `afficherCase(...)` qui affiche la case en `(x,y)` d'un `Echiquier`.

(1 point) Enfin écrivez une opération `afficherEchiquier(...)` qui affiche un `Echiquier` sous la forme suivante (cas d'un échiquier de dimension 5 avec 3 joueurs après initialisation) :

```
x012345
y******
1*..2..
2*.3...
3*.....
4*....1
5*.....
```

Validez vos opérations avec la solution.

Solution C++ @[UtilsEchiquier.cpp]

```
/**
  Affiche une case
  @param[in] x - colonne de la case
  @param[in] y - ligne de la case
  @param[in] valeur - valeur de la case
*/
void afficherCase(int x, int y, int valeur)
{
  gotoxy(x+OFFSET_X, y+OFFSET_Y);
  if (valeur > 0)
  {
     cout<<valeur;
  }
  else if (valeur == VIDE)
```

```
  {
     cout<<'.';
  }
  else if (valeur == BORD)
  {
     cout<<'*';
```

```
  else
  {
     cout<<'#';
  }
}

/**
  Affiche un Echiquier
  @param[in] cb - un Echiquier
*/
void afficherEchiquier(const Echiquier& cb)
{
  clrscr();
  gotoxy(OFFSET_X-1, OFFSET_Y-1);
  cout<<"x";
  for (int x = 1; x <= cb.ndim; ++x)
  {
    gotoxy(OFFSET_X-1, OFFSET_Y+x);
    cout<<x%10;
  }
  gotoxy(OFFSET_X-1, OFFSET_Y);
  cout<<"y";
  for (int y = 0; y <= cb.ndim; ++y) // ligne
  {
    gotoxy(OFFSET_X+y, OFFSET_Y-1);
    cout<<y%10;
    for (int x = 0; x <= cb.ndim; ++x) // colonne
    {
       afficherCase(x,y,evalCase(cb,x,y));
    }
  }
}
```

Type Position

Une case est caractérisée par ses coordonnées : indice de ligne et indice de colonne.

(0.25 point) Définissez le type `Position` modélisant la position d'une case.

(0.25 point) Écrivez une opération `caseDansEchiquier(...)` qui teste et renvoie `Vrai` si une case en une `Position` donnée est dans le plateau d'un `Echiquier`, `Faux` sinon.

(0.25 point) De même, écrivez une opération `caseVide(...)` qui teste et renvoie `Vrai` si une case en une `Position` donnée (de coordonnées valides) d'un `Echiquier` est `VIDE`, `Faux` sinon.

Validez votre définition et vos opérations avec la solution.

Solution C++ @[UtilsEchiquier.cpp]

```
/** Type Position (d'une case) */
struct Position
{
   int x; //! numero de colonne
   int y; //! numero de ligne
};
```

```
/**
   Teste si une case est dans un Echiquier
   @param[in] cb - un Echiquier
   @param[in] pos - Position de la case
   @return Vrai si pos est dans le plateau effectif de cb
*/
bool caseDansEchiquier(const Echiquier& cb, const Position& pos)
{
   return (1 <= pos.x and pos.x <= cb.ndim and 1 <= pos.y and pos.y <= cb.ndim);
}
```

```
/**
   Predicat de case VIDE d'un Echiquier
   @param[in] cb - un Echiquier
   @param[in] pos - Position de la case
   @return Vrai si pos est une case VIDE
   @pre pos est dans l'Echiquier
*/
bool caseVide(const Echiquier& cb, const Position& pos)
{
   return evalCase(cb, pos.x, pos.y) == VIDE;
}
```

2.2 Modélisation des joueurs

Analyse
La caractéristique d'un joueur est son score (nombre de points).

Définissez la constante `NJOUEURS=10` (nombre maximal de joueurs), puis le type `Joueurs` comme étant un tableau d'entiers mémorisant les scores.

Écrivez une procédure `initialiserJoueurs(...)` qui initialise le tableau des points à zéro. Pour des raisons de cohérence, on pourra considérer que les joueurs sont numérotés à partir de 1.

Écrivez une procédure `actualiserPoints(...)` qui incrémente les points du joueur `k` de `npoints` dans un `Joueurs`.

Validez vos procédures avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/** Nombre Maximum de joueurs */
const int NJOUEURS = 10;
```

```
/** Type Joueurs: nombre de points */
typedef int Joueurs[NJOUEURS];
```

```
/**
  Initialise les joueurs
  @param[out] joueurs - un Joueurs
  @param[in] n - nombre de joueurs
*/
void initialiserJoueurs(Joueurs& joueurs, int n)
{
  for (int k = 1; k <= n; ++k)
  {
    joueurs[k] = 0;
  }
}
```

```
/**
  Actualise les points d'un joueur
  @param[in] k - numero du joueur
  @param[in] npoints - nombre de points
  @param[in,out] joueurs - un Joueurs
*/
void actualiserPoints(int k, int npoints, Joueurs& joueurs)
{
  joueurs[k] += npoints;
}
```

2.3 Découpage du problème

Donnez en français, en quoi consiste le **déroulement d'une partie**.

Solution simple

Dans la version joueur(s)/joueur, l'ordinateur est l'ordonnanceur du jeu : il initialise et affiche le jeu, lit la réponse validée d'un des joueurs (la nouvelle position du curseur), visualise le nouvel état puis teste si le joueur suivant peut encore jouer. Si oui, la partie continue, sinon elle est finie. Ensuite, il calcule et affiche le(s) gagnant(s), c.-à-d. celui qui a le plus de points.

En quoi consiste le **tour d'un joueur** ?

Solution simple
Celui-ci est composé de trois étapes :

1. L'affichage de la grille de jeu.
2. Le joueur actif donne la nouvelle position du curseur et incrémente son nombre de points.
3. Puis il donne la main au joueur suivant.

Quelles sont les **conditions d'arrêt** ?

Solution simple
L'arrêt de la partie revient à vérifier que le joueur actif ne peut plus prendre d'objets sur le plateau.

Découpage en sous-algorithmes
Il est résumé dans la figure ci-dessous. Les flèches indiquent la dépendance entre les algorithmes (quels algorithmes en utilisent d'autres). jeu.

2.4 Modélisation du Jeu

Analyse
Le jeu *TirsCroises* est composé d'un plateau, des points des joueurs, du nombre de joueurs au total, du nombre de joueurs machine (dans la version IA), de la position du curseur et du numéro du joueur actif.

Définissez le type `TirsCroises` modélisant le jeu.

Écrivez une procédure `afficherJeu(...)` qui affiche un `TirsCroises` en affichant son plateau, le numéro du joueur actif, son nombre de points ainsi que la position du curseur.

Écrivez une procédure `donnerMainSuivant(...)` qui donne la main au joueur suivant.

Validez votre définition et vos procédures avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/** Type Jeu */
struct JeuTirs
{
   Echiquier plateau; //! plateau du jeu
   Joueurs joueurs; //! points des joueurs
   int njoueurs; //! nombre de joueurs (au total)
   int njmachines; //! nombre de joueurs machine
   Position curseur; //! position du curseur
```

```
   int k; //! numero du joueur actif
};
```

```
/**
   Affiche un Jeu
*/
void afficherJeu(const JeuTirs& jeu)
{
   // Affiche le plateau
   afficherEchiquier(jeu.plateau);
   // Affiche le joueur actif
   const int taille = jeu.plateau.ndim;
   gotoxy(1, taille+3+OFFSET_Y);
   cout<<(jeu.k <= jeu.njoueurs-jeu.njmachines ? "Joueur actif: " : "Machine active:
    ")<<jeu.k;
   cout<<" npoints: "<<jeu.joueurs[jeu.k];
   // Affiche la position du curseur
   gotoxy(1, taille+4+OFFSET_Y);
   cout<<"Curseur: col= "<<jeu.curseur.x<<" lig= "<<jeu.curseur.y<<endl;
}
```

```
/**
  Donne la main au joueur suivant
  @param[in,out] jeu - un JeuTirs
*/
void donnerMainSuivant(JeuTirs& jeu)
{
  if (++jeu.k > jeu.njoueurs)
  {
    jeu.k = 1;
  }
}
```

...(suite page suivante)...

3 Version Joueur(s)/Joueur

3.1 Initialisation du jeu (1 point)

Analyse

La procédure d'initialisation du jeu devra :

- Demander et saisir la taille de l'échiquier.
- Initialiser l'échiquier.
- Initialiser le nombre de joueur machine.
- Demander et saisir le nombre total de joueurs.
- Initialiser les points des joueurs.
- Placer au hasard le curseur.
- Tirer au hasard le joueur qui commence.

Écrivez une procédure `initialiserJeu(...)` qui initialise un `JeuTirs jeu`.

Validez votre procédure avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Entier pseudo-aleatoire
  @param[in] n - un entier
  @return un entier pseudo-aleatoire dans [0..n-1]
*/
int aleatoire(int n)
{
  return (rand()%n);
}
```

```
/**
  Initialise un JeuTirs (echiquier, joueurs, curseur)
  @param[out] jeu - un JeuTirs
  @param[in] njmachines - nombre de joueurs machine
*/
void initialiserJeu(JeuTirs& jeu, int njmachines=0)
{
  int taille;
  cout<<"Taille de l'echiquier: ";
  taille = saisieEntier(NMIN, NMAX-1);
  initialiserEchiquier(jeu.plateau, taille);
  for (int y = 1; y <= taille; ++y)
  {
     for (int x = 1; x <= taille; ++x)
     {
        fixerCase(jeu.plateau,x,y,aleatoire(9)+1);
     }
  }
```

```
  jeu.njmachines = njmachines;
  cout<<"Nombre de joueurs machine = "<<jeu.njmachines<<endl;
  cout<<"Nombre total de joueurs? ";
  jeu.njoueurs = saisieEntier(2, NJOUEURS-1);
  initialiserJoueurs(jeu.joueurs, jeu.njoueurs);

  jeu.curseur.x = aleatoire(taille)+1;
  jeu.curseur.y = aleatoire(taille)+1;
  fixerCase(jeu.plateau, jeu.curseur.x, jeu.curseur.y, CURSEUR);

  jeu.k = aleatoire(jeu.njoueurs) + 1;
}
```

Écrivez une procédure `test_initialisationJJ` qui déclare un `JeuTirs`, l'initialise puis l'affiche.

Outil C++

L'initialisation du générateur de nombres pseudo-aléatoires (ici avec l'horloge système) s'effectuera dans le **programme principal** avec l'instruction :

```
srand(time(NULL));
```

La procédure `srand` est définie dans la bibliothèque `<random>` et la fonction `time` dans la bibliothèque `<ctime>`.

Testez.

Validez votre procédure avec la solution.

Solution C++ @[pgtirscroises1.cpp]

```
void test_initialisationJJ()
{
  JeuTirs jeu;
  initialiserJeu(jeu);
  afficherJeu(jeu);
}
```

3.2 Destruction d'une case (2.5 points)

Analyse

Une case est valide si :

1. Elle est **dans** le plateau de l'échiquier.
2. **Et** elle est sur la ligne **ou** colonne du curseur.
3. **Et** elle est de valeur positive.

(0.5 point) Écrivez une fonction `caseValide(...)` qui teste et renvoie `Vrai` si une case en une `Position` donnée d'un `JeuTirs` est valide, `Faux` sinon.

(0.5 point) Déduisez une fonction `demanderCaseADetruire(...)` qui demande au joueur actif la `Position` de la case à prendre **jusqu'à** ce qu'elle soit valide puis renvoie cette position.

(0.5 point) Écrivez une procédure `detruireCase(...)` qui détruit la case en une `Position` donnée (coordonnées valides) d'un `JeuTirs`.

Aide simple

La case du curseur devient une case vide et la case à détruire devient la nouvelle position du curseur.

(1 point) Enfin écrivez une procédure `detruireCaseSurJeuJJ(...)` qui demande au joueur actif la case à détruire, actualise son nombre de points, puis détruit la case sur un `JeuTirs`.

Validez vos fonctions et procédures avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Teste si une case d'un Jeu est valide
  (<=> dans l'echiquier, sur ligne ou colonne du curseur et non vide)
  @param[in] jeu - un JeuTirs
  @param[in] pos - Position de la case
  @return Vrai si pos est valide
*/
bool caseValide(const JeuTirs& jeu, const Position& pos)
{
  return caseDansEchiquier(jeu.plateau, pos)
  and (pos.x == jeu.curseur.x or pos.y == jeu.curseur.y)
  and evalCase(jeu.plateau, pos.x, pos.y) > 0;
}
```

```
/**
  Demande une case a detruire jusqu'a ce qu'elle soit valide
  @param[in] jeu - un Jeu
  @return la Position de la case a detruire
*/
Position demanderCaseADetruire(const JeuTirs& jeu)
{
   Position dest;
   do{
      gotoxy(1, jeu.plateau.ndim+5+OFFSET_Y);
      cout<<"Case a detruire x y? ";
      cin>>dest.x>>dest.y;
   } while (not caseValide(jeu,dest));
   return dest;
}
```

```
/**
  Detruit une case d'un Jeu
  @param[in,out] jeu - un JeuTirs
  @param[in] pos - Position de la case a detruire
*/
void detruireCase(JeuTirs& jeu, const Position& pos)
{
  fixerCase(jeu.plateau, jeu.curseur.x, jeu.curseur.y, VIDE);
  jeu.curseur = pos;
  fixerCase(jeu.plateau, pos.x, pos.y, CURSEUR);
}
```

```
/**
  Destruction d'une case par le joueur actif (Joueur/Joueur(s))
  @param[in,out] jeu - un Jeu
*/
void detruireCaseSurJeuJJ(JeuTirs& jeu)
{
  Position tir = demanderCaseADetruire(jeu);
  actualiserPoints(jeu.k, evalCase(jeu.plateau, tir.x, tir.y), jeu.joueurs);
  detruireCase(jeu, tir);
}
```

3.3 Test de partie terminée

Analyse

Une partie est terminée s'il n'y a plus de case à prendre sur la ligne ou colonne du curseur.

Écrivez une fonction `partieTerminee(...)` qui teste et renvoie `Vrai` si la partie est terminée, `Faux` sinon.

Aide simple

Traitez la négative : il suffit d'une case ayant une valeur positive pour que la partie ne soit pas terminée.

Validez votre fonction avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Teste si une partie est terminee
  @param[in,out] jeu - un Jeu
  @return Vrai si la partie est terminee
*/
bool partieTerminee(const JeuTirs& jeu)
{
   const Echiquier &cb = jeu.plateau;
   bool rs = false;
   for (int k = 1; k <= cb.ndim && not rs; ++k)
```

```
  {
    rs = evalCase(cb, k, jeu.curseur.y) > 0;
  }
  for (int k = 1; k <= cb.ndim && not rs; ++k)
  {
    rs = evalCase(cb, jeu.curseur.x, k) > 0;
  }
  return not rs;
}
```

3.4 Affichage des gagnants

Écrivez une procédure `afficherGagnant(...)` qui affiche les scores des `Joueurs` ainsi que le(s) joueur(s) gagnant(s).

Validez votre procédure avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Affiche les points des joueurs ainsi que le(s) gagnant(s)
  @param[in] jeu - Un jeuTirs
*/
void afficherGagnant(const JeuTirs& jeu)
{
  gotoxy(1, jeu.plateau.ndim+6+OFFSET_Y);
  const Joueurs &joueurs = jeu.joueurs;
  int ngagnants = 0;
  int gagnant = 1;
  for (int k = 1; k <= jeu.njoueurs; ++k)
  {
    cout<<"Joueur "<<k<<" npoints = "<<joueurs[k]<<endl;
    if (joueurs[k] > joueurs[gagnant])
    {
      gagnant = k;
    }
    else if (joueurs[k] == joueurs[gagnant])
    {
      ++ngagnants;
    }
  }
  if (ngagnants == 1)
  {
    cout<<"Le JOUEUR "<<gagnant<<" A GAGNE"<<endl;
  }
  else for (unsigned k = 1; k <= jeu.njoueurs; ++k)
  {
    if (joueurs[k] == joueurs[gagnant])
    {
      cout<<"Le JOUEUR "<<k<<" A GAGNE"<<endl;
    }
  }
}
```

3.5 Mise en place du tout (1.5 points)

(1.5 points) Écrivez une procédure `jeuTirsCroisesJJ` qui effectue une partie Joueur(s) contre Joueur. La procédure devra :

- Déclarer un `JeuTirs` et l'initialiser.
- Itérez `TantQue` la partie n'est pas terminée :
  - Afficher le jeu.
  - Afficher le joueur actif et son nombre de points.
  - Afficher la position du curseur (pour mémo).
  - Demander la case à détruire au joueur actif.
  - Donner la main au joueur suivant.
- Afficher les scores et le(s) gagnant(s).

Validez votre procédure avec la solution.

Solution C++ @[pgtirscroises1.cpp]

```
void jeuTirsCroisesJJ()
{
  cout<<"Jeu Joueur(s) contre Joueur"<<endl;
  // Initialise le jeu (echiquier, joueurs)
  JeuTirs jeu;
  initialiserJeu(jeu);

  // Taille du plateau
  const int taille = jeu.plateau.ndim;
  // Itere tantque la partie n'est pas terminee
  while (not partieTerminee(jeu))
  {
    // Affiche le jeu
    afficherJeu(jeu);

    // Le joueur actif detruit une case
    detruireCaseSurJeuJJ(jeu);
    afficherJeu(jeu);

    // Donne la main au joueur suivant
    donnerMainSuivant(jeu);
  }
  afficherGagnant(jeu);
}
```

...(suite page suivante)...

4 Version Joueur(s)/Machine

4.1 Stratégies de la machine

Dans la version solitaire, la machine joue l'adversaire.
Vous devrez définir les deux stratégies suivantes :

Écrivez une fonction `rechCaseADetruireJMnaif(...)` qui recherche et renvoie la `Position| de la case à détruire dans un \lstinline`JeuTirs@. Dans cette version, la machine recherche une case de valeur maximale dans la colonne ou ligne du curseur : stratégie très naïve.

Écrivez une fonction `rechCaseADetruireJMniv1(...)` qui recherche et renvoie la `Position` de la case à détruire dans un `JeuTirs`. Dans cette version, la machine se positionne sur chacune des positions jouables et recherche la valeur maximale que peut jouer l'adversaire en supposant qu'il joue naïvement la plus grande des valeurs issue de cette position. Ensuite, il calcule la (une) position qui offre le plus grand score, c.-à-d. une position où la différence entre la valeur jouable et celle de l'adversaire est la plus grande : stratégie de profondeur 1.

Aide simple
Il est utile de disposer d'une structure de données `ListePos` qui mémorise la liste des positions jouables issues d'une position (colonne, ligne).

Validez vos fonctions avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Recherche la case a detruire sur un jeu (strategie naive)
  Ne peut faillir puisque l'on peut jouer
  @param[in] jeu - un Jeu
  @return la Position de la case a detruire
*/
Position rechCaseADetruireJMnaif(const JeuTirs& jeu)
{
  ListePos ls;
  // Liste des positions jouables issue de celle du curseur
  rechPosJouables(jeu.plateau, jeu.curseur.x, jeu.curseur.y, ls);
  // Recherche la position qui offre le plus grand score
  int kmax = 0;
  int dmax = evalCase(jeu.plateau, ls.x[kmax], ls.y[kmax]);
  for (int k = 1; k < ls.taille; ++k)
  {
    int val = evalCase(jeu.plateau, ls.x[k], ls.y[k]);
    if (dmax < val)
    {
      kmax = k;
      dmax = val;
    }
```

```
  }
  Position tir;
  tir.x = ls.x[kmax];
  tir.y = ls.y[kmax];
  return tir;
}
```

```
/**
  Recherche la case a detruire sur un jeu (profondeur 1)
  Se positionne sur chacune des positions jouables et recherche la
  valeur maximale que peut jouer l'adversaire en supposant qu'il joue
  naivement la plus grande des valeurs issue de cette position.
  Calcule la (une) position qui offre le plus grand score (i.e. position
  ou la difference entre la valeur jouable et celle de l'adversaire
  est la plus grande).
  @param[in] jeu - un Jeu
  @return la Position de la case a detruire
*/
Position rechCaseADetruireJMniv1(const JeuTirs& jeu)
{
  ListePos ls;
  // Liste des positions jouables issue de celle du curseur
  rechPosJouables(jeu.plateau, jeu.curseur.x, jeu.curseur.y, ls);
  // Recherche la position qui offre le plus grand score
  int kmax;
  int dmax = NULL_DIFF;
  for (int k = 0; k < ls.taille; ++k)
  {
    ListePos adverse;
    // Position jouables de l'adversaire issue de la position courante
    rechPosJouables(jeu.plateau, ls.x[k], ls.y[k], adverse);
    // Determine la valeur (sensee être) jouee par l'adversaire
    int vmax = 0;
    for (int j = 0; j < adverse.taille; ++j)
    {
      int val = evalCase(jeu.plateau, adverse.x[j], adverse.y[j]);
      if (not posIdentiques(ls.x[k], ls.y[k], adverse.x[j], adverse.y[j]) and vmax < val)
      {
        vmax = val;
      }
    }
    // Actualise la position, si meilleure
    int diff = evalCase(jeu.plateau, ls.x[k], ls.y[k]) - vmax;
    if (dmax < diff)
    {
      dmax = diff;
      kmax = k;
    }
  }
  Position tir;
  tir.x = ls.x[kmax];
  tir.y = ls.y[kmax];
  return tir;
}
```

4.2 Destruction d'une case

Faites un copier/coller de la procédure `detruireCaseSurJeuJJ` en la procédure `detruireCaseSurJeuJM(...)` qui demande au joueur actif la `Position` de la case à détruire puis la détruit sur un `JeuTirs` comprenant des joueurs machine.

Aide simple

Modifiez la procédure de sorte que :

- Si le joueur actif est un humain : faites-le jouer comme dans la version joueur(s)/joueur.
- Sinon dans le cas de la machine : déterminez la case à détruire en utilisant une des stratégies de la machine.

Validez votre procédure avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Destruction d'une case par le joueur actif sur un jeu Joueurs/Machine
  @param[in,out] jeu - un JeuTirs
*/
void detruireCaseSurJeuJM(JeuTirs& jeu)
{
  Position tir;
  if (jeu.k <= jeu.njoueurs - jeu.njmachines)
  {
    tir = demanderCaseADetruire(jeu);
  }
  else
  {
    tir = rechCaseADetruireJM(jeu);
    gotoxy(1, jeu.plateau.ndim+4+OFFSET_Y);
    cout<<"Machine va detruire "<<tir.x<<" "<<tir.y<<endl;
    system("Pause");
  }
  actualiserPoints(jeu.k, evalCase(jeu.plateau, tir.x, tir.y), jeu.joueurs);
  detruireCase(jeu, tir);
}

}
```

4.3 Procédure jeuTirsCroisesJM

Écrivez une procédure `jeuTirsCroisesJM` qui effectue une partie Joueur(s) contre Machine.

Validez votre procédure avec la solution.

Solution C++ @[pgtirscroises1.cpp]

```
void jeuTirsCroisesJM()
{
  cout<<"Jeu Joueur(s) contre Machine"<<endl;
  JeuTirs jeu;
  initialiserJeu(jeu, 1);

  // Taille du plateau
  const int taille = jeu.plateau.ndim;
  // Itere tantque la partie n'est pas terminee
  while (not partieTerminee(jeu))
  {
    // Affiche le jeu
    afficherJeu(jeu);

    // Le joueur actif detruit une case
    detruireCaseSurJeuJM(jeu);
    afficherJeu(jeu);

    // Donne la main au joueur suivant
    donnerMainSuivant(jeu);
  }
  afficherGagnant(jeu);
}
```

4.4 Stratégie de profondeur 2

On souhaite réaliser une stratégie de profondeur 2 pour la machine.

Écrivez une fonction `rechCaseADetruireJM(...)` recherche et renvoie la `Position| de la case à détruire dans un \lstinline`JeuTirs@. Dans cette version, la machine se positionne sur chacune des positions jouables et recherche la valeur maximale que peut jouer l'adversaire en supposant qu'il joue naïvement la plus grande des valeurs issue de cette position. Puis pour ces positions, il itère l'opération de recherche de la position jouable pour la machine de plus grand score. Enfin il effectue le cumul des différences et restitue la (une) position qui offre le plus grand score pour la machine.

Que pensez-vous de cette stratégie ?
Comment pouvez-vous l'améliorer ?

Aide simple
En jouant un peu, on constate rapidement que la tactique de l'humain n'est pas de jouer naïvement la case qui offre le plus grand gain.

Validez votre procédure avec la solution.

Solution C++ @[UtilsTirsCroises.cpp]

```
/**
  Recherche la case a detruire sur un jeu (profondeur 2)
  Se positionne sur chacune des positions jouables et recherche la
  valeur maximale que peut jouer l'adversaire en supposant qu'il joue
  naivement la plus grande des valeurs issue de cette position.
  Puis pour ces positions, itère l'operation de recherche de la position
  jouable pour la machine de plus grand score. Enfin effectue le cumul
  des differences et restitue la (une) position qui offre le plus grand
  score pour la machine.
  @param[in] jeu - un Jeu
  @return la Position de la case a detruire
*/
Position rechCaseADetruireJM(const JeuTirs& jeu)
{
  ListePos ls;
  // Liste des positions jouables issue de celle du curseur
  rechPosJouables(jeu.plateau, jeu.curseur.x, jeu.curseur.y, ls);
  // Recherche la position qui offre le plus grand score
  int kmax;
  int dmax = NULL_DIFF;
  for (int k = 0; k < ls.taille; ++k)
  {
    ListePos adverse;
    // Position jouables de l'adversaire issue de la position courante
    rechPosJouables(jeu.plateau, ls.x[k], ls.y[k], adverse);
    // Determine la valeur (sensee être) jouee par l'adversaire
    int vmax = 0;
    for (int j = 0; j < adverse.taille; ++j)
    {
      int val = evalCase(jeu.plateau, adverse.x[j], adverse.y[j]);
      if (not posIdentiques(ls.x[k], ls.y[k], adverse.x[j], adverse.y[j]) && vmax < val)
      {
        vmax = val;
      }
    }
    // Epure les positions de valeur non maximale
    int j = 0;
    while (j < adverse.taille)
    {
      int val = evalCase(jeu.plateau, adverse.x[j], adverse.y[j]);
      if (posIdentiques(ls.x[k], ls.y[k], adverse.x[j], adverse.y[j]) or val != vmax)
      {
        retirerListe(adverse, j);
      }
      else
      {
        j += 1;
      }
    }
    // Si l'adversaire ne peut plus jouer: la machine dispose de la
    // seule position en k. Sinon recherche pour chacune des positions
    // de valeur maximale jouables par l'adversaire, la valeur maximale
    // jouable par le joueur (machine)
    if (adverse.taille == 0)
    {
      int diff = evalCase(jeu.plateau, ls.x[k], ls.y[k]);
      if (dmax < diff)
```

```
      {
        dmax = diff;
        kmax = k;
      }
    }
    else
    {
      for (int j = 0; j < adverse.taille; ++j)
      {
        ListePos joueur;
        rechPosJouables(jeu.plateau, adverse.x[j], adverse.y[j], joueur);
        int vmax = 0;
        for (int jj = 0; jj < joueur.taille; ++jj)
        {
          if (not posIdentiques(ls.x[k], ls.y[k], joueur.x[jj], joueur.y[jj])
          and not posIdentiques(adverse.x[j], adverse.y[j], joueur.x[jj], joueur.y[jj]))
           {
             int val = evalCase(jeu.plateau, joueur.x[jj], joueur.y[jj]);
             if (vmax < val)
             {
               vmax = val;
             }
           }
        }
        int diff = evalCase(jeu.plateau, ls.x[k], ls.y[k]) - evalCase(jeu.plateau,
    adverse.x[j], adverse.y[j]) + vmax;
        if (dmax < diff)
        {
          dmax = diff;
          kmax = k;
        }
      }
    }
  }
  Position tir;
  tir.x = ls.x[kmax];
  tir.y = ls.y[kmax];
  return tir;
}
```

5 Références générales

Comprend [Jeux et Stratégies :n7] ■