Algorithmique — C++ — Théorie des Jeux

1 Extensions de base

1.1 Commentaires

```
/* commentaire C. avec des  // /// //
   sur plusieurs lignes */
// Commentaire C++, jusqu'à /* la fin */ de la ligne.
```

1.2 & et const

Esperluète : & (ampersand) ≡ alias sur une variable ou un objet.

```
int  ix;
int& ir = ix; // ref sur ix
ix = 1;       // ir aussi
ir = 2;       // ix aussi
```

Passage de paramètres

```
void e(      T  i ) ;  // i est copié, la copie est utilisée
                       // par la fonction.
void f(      T& i ) ;  // les modifications sur i seront
                       // valables globalement.
void g(const T  i ) ;  // i est copié, la copie n'est utilisée
                       // qu'en lecture.
void h(const T& i ) ;  // i ne peut pas être modifié.
```

Déclarations constantes

```
int i;                          // Entier ordinaire
const int i = 13;               // Entier non modifiable
int * ip;                       // Pointeur
int * const cp = &i;            // Pointeur non modifiable
const int * cip;                // Pointeur sur un entier non modifiable
const int * const cicp = &ci; // Pointeur non modifiable
                                // sur un entier non modifiable
```

Surcharge des fonctions

```
int max(int i, int j) { return (i>j) ? i : j; }
char* abs(char * s, char * t) {
  return (strcmp(s,t)>0) ? s : t; }
```

2 Classes

Définition d'un nouveau *type* C++ : ensemble de *membres* (données) et de *méthodes* (fonctionnalités).

```
class Point {
   int _x, _y;
public: // Interface utilisateur
   int abscisse()  { return _x; }
   int ordonnee() { return _y; }
   void affect_x(int x) { _x = x; }

};
```

Une classe est un type. Une variable de ce type (une instance de la classe) est appelée un *objet*.

2.1 public, private, protected

struct (accès public par défaut) $\approx$ class (accès privé par défaut).

Troisième accès : protected $\rightarrow$ autorisé aux classes dérivées.

2.2 Constructeurs

Utilisés à la déclaration de l'objet pour le construire (affecter les membres, allouer la mémoire, etc.) à partir de différents paramètres.

```
class Point {
   int _x, _y;
public: // Interface utilisateur

   Point() : _x(0), _y(0) { }
   Point(int x, int y) : _x(x), _y(y) { }
   Point(const Point& P) : _x(P._x), _y(P._y) { }

   int abscisse()  { return _x; }
   int ordonnee() ;

};

Point X;      // appel du constructeur vide
Point M(3,4); // appel du constructeur sur les entiers
Point P(M);   // appel du constructeur de recopie physique
```

Par défaut, une classe implémente le constructeur vide et le constructeur de recopie par des appels aux constructeurs de ses membres.

2.3 Destructeur

Appelé automatiquement quand l'objet ne sera plus utilisé (fermeture d'accolade }).

```
class Tableau {
   int * _d;
public:
   Tableau(int s = 10) { _d = new int[s]; }
   ~Tableau() { delete [] _d; }

};

main() {
  {  Tableau t; // de taille 10 par défaut
     ...
  }            // les destructeurs de tous les objets
               // déclarés dans ce groupe sont appelés.
}
```

△ Par défaut, une classe implémente un destructeur qui ne fait rien.

Constructeurs et Destructeur ⇒ Intérêt majeur de la programmation objet : la gestion de la mémoire est faite *par le programmeur une fois pour toute* et non *par l'utilisateur à chaque utilisation* !

3 Optimisation par annotations

3.1 const

Pour permettre au compilateur de mieux optimiser le code, le mot-clé `const` peut être ajouté à une méthode pour indiquer que celle-ci ne modifiera pas les membres de la classe.

```
class Point {
   int _x, _y;
public: // Interface utilisateur
   int abscisse() const { return _x; }
   void affect_x(int x)  { _x = x; }

};
```

3.2 :: et inline

Il est possible de définir une méthode en dehors de la définition de la classe : il suffit de préciser la classe pour laquelle la méthode déclarée doit être implémentée.

```
class Point {
   int _x, _y;
public: // Interface utilisateur
   Point();
   int abscisse() const;
   void affect_x(int);
};

Point::Point() : _x(0), _y(0) {  }
int Point::abscisse() const { return _x; }
void Point::affect_x(int x) { _x = x; }
```

Les méthodes définies à l'intérieur de la classe (méthodes *inline*) peuvent avoir leur corps recopié à chaque appel pour accélérer l'exécution. Ce n'est pas le cas de celle définies à l'extérieur.

Une méthode définie à l'extérieur, comme toute fonction, peut aussi être déclarée *inline*. Le compilateur ESSAIERA de recopier le corps si le code n'est pas trop important.

```
inline Point::Point() : _x(0), _y(0) {  }
inline int Point::abscisse() const { return _x; }
inline void Point::affect_x(int x) { _x = x; }
inline void f(int i) { return 2*i+1; }
```

4 this

Dans une méthode, ou un constructeur, le mot-clé `this` est un pointeur sur l'objet considéré.

```
Point& Point::copie(const Point& P) {
  if (this == &P) return *this; // ce n'est pas la peine de
                                // se copier soit même.
    // Attention, ce test vérifie l'égalité en adresse mémoire
    // et non l'égalité mathématique (celle des membres).
  _x = P._x;
  this->_y = P._y;
}
```

5 static

Le mot-clé static permet de définir des membres ou des méthodes «globales».

Un membre static est identique pour toutes les instances de la classe. Toute modification par un objet est reconnue par tous les autres objets.

Une méthode est normalement appelée au travers d'un objet. Une méthode static est appelée par la classe (pour manipuler les membres static).

```
class Repere {
  static Point _origine;
public:
  Repere() : _origine(0,0) {}
  static void changement_origine(const Point& P) {
    _origine = P;
  }
  Point origine() { return _origine; }
};

main() {
   Repere R1, R2;
   Point M(3,4);
   Repere::changement_origine( M );
   printf(``Abscisse de l'origine: %l'', (R2.origine()).abscisse() );
}
```

6 friend

Une méthode friend est en fait une fonction classique, mais déclarée à l'intérieur d'une classe (et précédée du mot-clé friend). Cela lui permet de manipuler les membres ou méthodes privées de la classe.

De même, une classe peut être déclarée à l'intérieur d'une autre classe, précédée du mot-clé friend). Ainsi les méthodes de cette classe (implémentées ultérieurement) pourront manipuler les données privées de la classe dont elle est «amie».

7 Opérateurs

Un opérateur est une fonction ou une méthode avec un appel particulier.

7.1 Exemple

```
class Complexe {
  float _re, _im;
public:
    // une méthode peut être un operateur ...
  Complexe& operator+= (Complexe x) {
    _re += x._re;
    _im += x._im;
    return *this;
  }
};

  // ... tout comme une fonction classique
Complexe operator+ (Complexe x, Complexe y) {
  Complexe r = a;
  return r += b;
}

void f(Complexe x, Complexe y, Complexe z) {
  Complexe r1 = x + y + z; // r1 = operator+(x, operator+(y,z) )
  Complexe r2 = x;
  r2 += y;                 // r2.operator+=( y )
}
```

△ l'opérateur d'affectation (`operator=`) est prédéfini et appelle par défaut les opérateurs d'affectation des membres de l'objet. Il peut être redéfini.

7.2 Opérateur parenthèses

L'opérateur parenthèses permet de donner un type à une fonction (de définir une fonction-classe). Ainsi les fonctions peuvent être manipulées plus facilement (comme des objets).

```
class Incrementeur {
  int _inc;
public:
  Incrementeur(int i) : _inc(i) {}
  int operator() (int i) { return i+_inc; }
};

main() {
  Incrementeur plus_un(1), plus_deux(2);

  int a = plus_deux(3);  // a <-- 5;
}
```

7.3 Opérateurs d'incrémentation et de décrémentation

operator++ et operator−− sont ambigus : s'agit-il de la définition le l'opérateur suffixé ou préfixé ? Un paramètre artificiel `int` dans la définition permet de lever cette ambiguïté.

```
class Pointeur {
  int * p;
public:
  Pointeur& operator++ ();    // préfixe
  Pointeur& operator++ (int); // suffixe
};
```

8 Entrées sorties par flots

La librairie standard <iostream.h> permet de faire des entrées-sorties par flots. Les flots de sorties (de type ostream) sont cout et cerr pour le standard et l'erreur respectivement. Le flot d'entrée (de type istream) est cin.

```
int numero;
cout << ``Entrez un nombre plus petit que 10 : ``;
cin >> numero;
if (numero > 10)
  cerr << ``\n *** Erreur votre nombre est trop grand !!!'' << endl;
else
  cout << ``\n Merci.'' << endl;
```

Il est possible de définir les opérateurs << et >> pour n'importe quel type et ainsi chaque objet peut être facilement manipulé par les flots.

```
class Complexe {
  float _re, _im;
public:
  ...

friend ostream& operator<< (ostream& o, const Complexe& c) {
  return o << c._re << ``+'' << c._im << ``i'';
}

friend istream& operator>> (istream& i, Complexe& c) {
  return i >> c._re >> c._im;
}
```

△ chaque << est un appel de fonction. Ces opérateurs sont donc commodes mais moins efficaces que les entrées-sorties C (scanf, printf) qui peuvent ne faire qu'un seul appel pour plusieurs objets.

9 Généricité

Une classe ou une fonction peut être générique par rapport à un type ou une valeur entière. La déclaration de la classe doit alors être préfixée par `template`.

9.1 Déclaration de classe générique

```
template <typename T, int taille_max = 200>
  // Le parametre entier max_size a la valeur 200 par defaut.
class  pile {
  private:
    T* la_pile ;
    int sommet ;

  public:
    typedef T value_type ;
    pile() : sommet(0) { la_pile = new T [taille_max] ; }
    ~pile() { delete [] la_pile ; }

    void empiler ( char c ) {
      la_pile[ sommet ] = c ;
      ++sommet ;
    }

    // ...
};

void main() {
  typedef pile< char, 26 > Alphabet;
  Alphabet p ; // instanciation d'une pile de 26 caracteres.
  for (Alphabet::value_type c = 'a'; c < 'z'; ++c) {
     p.empiler( c ) ; }
}
```

`template` permet aussi de déclarer des fonctions génériques, voire des fonctions membres génériques d'une classe générique.

10 Standard Template Library

La STL (Standard Template Library) fournit différents types génériques implémentant des containers d'objets. Un *container* peut être vu comme une séquence d'objets de type `T` qui peut être parcourue en utilisant des *itérateurs*.

10.1 Séquences

La STL propose trois séquences de base :

- `vector< T >` : vecteur à une dimension; permet l'accès rapide à n'importe quel élément en temps constant. L'insertion et la destruction d'un élément sont coûteuses.
- `list< T >` : liste doublement chainée. L'insertion et la destruction sont rapides; l'accès à un élément aléatoire coûteux.
- `deque< T >` : séquence pour laquelle l'ajout en tête ou en queue est aussi efficace que pour `list`, et où l'accès à n'importe quel élément est presque aussi efficace que pour un vecteur. L'insertion et la destruction en milieu sont coûteux.

Il existe trois containers de base optimisés, qui peuvent être vus comme des vecteurs :

- `basic_string` et `string` : chaîne de caractères;
- `bitset` : vecteur de bits (opérateurs logiques et de décalages);
- `valarray` : vecteur optimisé pour les calculs numériques (vectoriels).

Les méthodes communes à ces trois séquences sont les suivantes :

- size() : renvoie le nombre d'éléments
- resize(int i) : change le nombre d'éléments
- begin() délivre une référence sur le premier élément `C[0]`
- end() délivre une référence sur le premier élément qui n'appartient plus au container (juste après `C[N-1]`)
- front() le premier élément `*(C.begin())`
- back() le dernier élément `*(C.(--end()))`
- push_back(T x) $\equiv$ `a.insert(a.end(), x)`
- pop_back(T x) $\equiv$ `a.erase(--a.end())`

10.2 Itérateurs

De manière générale, un container `C` est une séquence de N objets de type `T : { C[0], ..., C[N-1] }`. Tout container contient des itérateurs qui permettent de le parcourir, en renvoyant l'équivalent de pointeurs sur ses éléments.

- `C.begin()` délivre une référence sur le premier élément `C[0]`;
- `C.end()` délivre une référence sur le premier élément qui n'appartient plus au container (juste après `C[N-1]`);

Trois principaux itérateurs sont proposés :

- `Forward iterator  : C.begin(), C.end().`
- `Backward iterator : C.rbegin(), C.rend().`
- `Random iterator   : C[i]` (sans vérification de débordement) ou `C.at(i)` (avec vérification).

```
#include <iostream>
#include <vector>
#include <list>

template<class T, template <class T> class Container>
ostream& operator<< (ostream& o, const Container<T>& C) {
  typename Container<T>::const_iterator refs =  C.begin();
  for(  ;  refs != C.end() ;  ++refs ) {
      o << *refs << " " ;
  }
  return o << endl;
}

main() {

  vector<int> v(2); v[0] = 1; v[1] = 2;
  list<char> l; l.push_back('f'); l.push_back('g');

  cout << "vecteur : " << v;
  cout << "list : " << l;
}
```

10.3 Algorithmes génériques

La STL fournit en standard différents algorithmes génériques qui peuvent être appliqués à de nombreux containers :

- `find, search, sort, for_each, copy, rotate, random_shuffle`, ...
- `min, max, lexicographical_compare,` ...
- opérations sur les ensembles : `includes, union, intersection,` ...
- construction d'une structure de tas : `make_heap, push_heap, pop_heap, sort_heap`
- Permutations : `next_permutation, prev_permutation`.

10.4 Piles et Files

Trois adapteurs sont fournis sur les containers de base `vector` et `deque` :

- `stack< T >` : pile LIFO
- `queue< T >` : file FIFO
- `priority_queue< T, cmp >` : file avec priorité.

Les attributs communs sont (la notion de "en tête" est relative au container : FIFO, LIFO ou plus prioritaire) :

- `push(const T& a)` : ajoute un élément
- `T& top()` : retourne l'élément en tête
- `pop()` : supprime l'élément en tête

Dans le cas d'une file de priorité, la priorité est définie par la fonction classe `cmp` qui soit avoir la spécification :

```
struct cmp{
  bool operator()( const T& x, const T& y) { ... }
};
```

Par défaut, la fonction classe `less` est utilisée. L'ordre des priorités est défini par rapport aux entiers. Sur les `int`, `less` implémente la comparaison au sens de l'opérateur `"<"` : l'élément le plus prioritaire est alors le plus grand.

10.5 Containers associatifs

Un container associatif est un container où chaque élément est associé à une clef (on dit aussi dictionnaire). Etant donné une clef, il est possible d'accéder à l'élément associé.

Un container associatif est générique par rapport à :
- le type `T` de ses éléments
- le type `Key` des clefs
- la fonction classe `Cmp` qui définit la comparaison entre deux clefs. Par défaut, la fonction classe `less` (correspond à l'opérateur `<`) est utilisée.

Les containers associatifs sont :
- `map< Key, T, Cmp > C` : à une clef correspond un unique élément (dictionnaire).
  `C[k]` retourne une référence sur l'objet de clef `k` (de type `Key`).
- `multimap< Key, T, Cmp > C` : à une clef peut correspondre plusieurs éléments.
- `set< Key, Cmp > C` : conserve uniquement les clefs (ne considère pas les éléments) qui sont ordonnées selon `Cmp`, pas les objets. Une clef apparaît au plus une fois. Un ensemble implémente donc un arbre équilibré de clefs. Il ne fournit pas l'opérateur `[]`.
- `multiset< Key, Cmp > C` : comme `set`, mais une même clef peut apparaître plusieurs fois.

11 Optimisations de compilation, Profiling

11.1 Options d'optimisation pour g++

Générales : `-O, -O3, -O5`

Non automatiques :

```
-funroll-loops             : déroule les boucles dont le nombre
d'itérations est connu à la compilation (moins de tests) ...

-funroll-all-loops         : toutes les boucles sont déroulées ...
                             (souvent plus lent que le précédent)

-frerun-loop-opt           : ... et redéroulées

-fexpensive-optimizations  : quelques optimisations mineures, qui
                             ralentissent la compilation

-felide-constructors      : Pas de temporaire créé pour initialiser
un objet (T x = y; est remplacé par T x(y);)
```

Spécifiques à la machine :

```
-mcpu=ultrasparc -mtune=ultrasparc  : sur Sun Ultra

-mcpu=rs6000 -mtune=rs6000          : sur IBM (ensibull)

-march=pentium; -march=pentiumpro   : sur PC intel
```

11.2 Profiling : gprof

1. Compiler votre programme avec l'option `-pg`
2. Faire une exécution de votre programme (`> ./mon_prog`)
3. Le fichier `gmon.out` contient les statistiques
4. Exécuter : `> gprof ./mon_prog > Stats.gmon`

Le fichier `Stats.gmon` contient des informations sur les fonctions appelées par votre programme : temps passé dans chaque fonction, nombre d'appels, etc.

Une première partie liste tous les appels de fonctions avec leur temps total. Une deuxième partie donne le détail des sous-fonctions appelées par chaque fonction.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 28.05     15.72    15.72 53155300     0.30     0.67  Plateau::UpdatePM(PossibleMove &, short, int, int, int, Position const &, short, bool &)
 14.56     23.88     8.16 68153700     0.12     0.64  Plateau::UpdatePM(PossibleMove &, short, int, int, bool &)
 14.26     31.87     7.99 24279986     0.33     0.43  vector<Position, allocator<Position> >::_M_insert_aux(Position *, Position const &)
  8.78     36.79     4.92  1362949     3.61    35.78  Plateau::UpdatePM(PossibleMove &, short, short)
  8.62     41.62     4.83    19771   244.30  2814.00  Plateau::ennemi_ab(int, double, double, short, short)
  4.59     44.19     2.57 48559972     0.05     0.05  __uninitialized_copy_aux__H2ZP8PositionZP8Position_X01X01X11G12__false_type_X11
  3.73     46.28     2.09 12722007     0.16     0.25  Coup::Coup(Coup const &)
  2.36     47.60     1.32 15437022     0.09     0.09  Position::BR(Position &) const
  2.32     48.90     1.30 15414874     0.08     0.08  Position::TL(Position &) const
  2.28     50.18     1.28 15836194     0.08     0.08  Position::TR(Position &) const
  2.23     51.43     1.25 15451775     0.08     0.08  Position::BL(Position &) const
  2.03     52.57     1.14  4137040     0.28     0.28  Plateau::unmove(vector<short, allocator<short> > const &, Coup const &, bool)
  1.98     53.68     1.11  4137040     0.27     0.31  Plateau::move(vector<short, allocator<short> > &, Coup const &, bool)
  1.91     54.75     1.07 12722315     0.08     0.08  __uninitialized_copy_aux__H2ZPC8PositionZP8Position_X01X01X11G12__false_type_X11
  1.12     55.38     0.63   651778     0.97     1.61  Plateau::addjump(PossibleMove &, short, Coup &, Position const &, int)
  0.59     55.71     0.33 10779582     0.03     0.03  __uninitialized_fill_n_aux__H3ZP8PositionZUiZ8Position_X01X11RCX21G12__false_type_X01
  0.29     55.87     0.16  4090653     0.04     0.04  vector<short, allocator<short> >::insert(short *, unsigned int, short const &)
  0.25     56.01     0.14  3532540     0.04     0.04  vector<Coup, allocator<Coup> >::insert(Coup *, unsigned int, Coup const &)
  0.05     56.04     0.03                             main
  0.00     56.04     0.00     2931     0.00     0.01  vector<Position, allocator<Position> >::operator=(vector<Position, allocator<Position> > const &)
  0.00     56.04     0.00     1721     0.00     0.00  Possibility::~Possibility(void)
  0.00     56.04     0.00     1528     0.00     0.37  __uninitialized_copy_aux__H2ZP4CoupZP4Coup_X01X01X11G12__false_type_X11
  0.00     56.04     0.00      992     0.00     0.01  Possibility::Possibility(Possibility const &)
  0.00     56.04     0.00      992     0.00     0.01  __uninitialized_copy_aux__H2ZPC4CoupZP4Coup_X01X01X11G12__false_type_X11
  0.00     56.04     0.00      764     0.00     0.99  vector<Coup, allocator<Coup> >::_M_insert_aux(Coup *, Coup const &)
  0.00     56.04     0.00      246     0.00     0.00  Timer::operator=(Timer const &)
  0.00     56.04     0.00      246     0.00     0.00  Timer::operator+(Timer const &) const

...
```

```
granularity: each sample hit covers 4 byte(s) for 0.20% of 4.95 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    4.95                 main [1]
                0.36    4.54    1768/1768        Plateau::ennemi_ab(int, double, double, short, short) [2]
                0.00    0.04    1083/100553      Plateau::UpdatePM(PossibleMove &, short, short) [3]
                0.00    0.00    2900/314904      Plateau::move(vector<short, allocator<short> > &, Coup const &, bool) [9]
                0.00    0.00    2900/314904      Plateau::unmove(vector<short, allocator<short> > const &, Coup const &, bool) [10]
                0.00    0.00     550/5028200     Plateau::UpdatePM(PossibleMove &, short, int, int, bool &) [4]
                0.00    0.00     172/1017564     Coup::Coup(Coup const &) [7]
                0.00    0.00      13/13          vector<PossibleMove, allocator<PossibleMove> >::insert(PossibleMove *, unsigned int, PossibleMove const &) [17]
                0.00    0.00      10/1927129     vector<Position, allocator<Position> >::_M_insert_aux(Position *, Position const &) [6]
                0.00    0.00       5/174         vector<Coup, allocator<Coup> >::_M_insert_aux(Coup *, Coup const &) [15]
                0.00    0.00     238/238         vector<Position, allocator<Position> >::operator=(vector<Position, allocator<Position> > const &) [22]
                0.00    0.00      10/313222      vector<short, allocator<short> >::insert(short *, unsigned int, short const &) [14]
                0.00    0.00     104/209         Possibility::~Possibility(void) [46]
                0.00    0.00      35/257902      vector<Coup, allocator<Coup> >::insert(Coup *, unsigned int, Coup const &) [23]
                0.00    0.00      22/22          Timer::operator+(Timer const &) const [48]
                0.00    0.00      22/22          Timer::operator=(Timer const &) [47]
                0.00    0.00      18/18          PossibleMove::PossibleMove(void) [51]
                0.00    0.00      17/17          vector<double, allocator<double> >::insert(double *, unsigned int, double const &) [24]
                0.00    0.00      14/14          vector<vector<short, allocator<short> >, allocator<vector<short, allocator<short> > > >::insert(vector<short, allocator<short> > *, unsigned int, vector<short, allocator<short> > const &) [25]
                0.00    0.00       1/1           Plateau::~Plateau(void) [54]
-----------------------------------------------
                              84719              Plateau::ennemi_ab(int, double, double, short, short) [2]
                0.36    4.54    1768/1768        main [1]
[2]     99.0    0.36    4.54    1768+84719   Plateau::ennemi_ab(int, double, double, short, short) [2]
                0.37    4.00   99470/100553      Plateau::UpdatePM(PossibleMove &, short, short) [3]
                0.09    0.02  312004/314904      Plateau::move(vector<short, allocator<short> > &, Coup const &, bool) [9]
                0.07    0.00  312004/314904      Plateau::unmove(vector<short, allocator<short> > const &, Coup const &, bool) [10]
                              84719              Plateau::ennemi_ab(int, double, double, short, short) [2]
-----------------------------------------------
...

-----------------------------------------------
                0.00    0.00      10/1927129     main [1]
                0.00    0.00   12359/1927129     Plateau::addjump(PossibleMove &, short, Coup &, Position const &, int) [13]
                0.74    0.11 1914760/1927129     Plateau::UpdatePM(PossibleMove &, short, int, int, int, Position const &, short, bool &) [5]
[6]     17.2    0.74    0.11 1927129         vector<Position, allocator<Position> >::_M_insert_aux(Position *, Position const &) [6]
                0.11    0.00 3854258/3854258     __uninitialized_copy_aux__H2ZP8PositionZP8Position_X01X01X11G12__false_type_X11 [8]
                0.00    0.00       3/20          __default_alloc_template<true, 0>::_S_refill(unsigned int) [50]
-----------------------------------------------

...
```

△ Ne pas faire `strip` avant l'exécution !
△ `_mcount` est une fonction interne à gprof.
△ Penser à désactiver les sorties d'erreur pour ne pas polluer les statistiques.
△ Si vos fonctions sont trop `inline` , elles peuvent ne pas apparaître, essayer de compiler avec les options suivantes :
`-fno-default-inline -fno-inline -finline-limit-0`

12 Arbres d'évaluation

12.1 Minimax

L'algorithme Minimax permet de choisir un "meilleur" successeur dans un arbre où chaque niveau est successivement à maximiser et à minimiser. L'algorithme parcourt l'arbre en profondeur d'abord.

```
function MINIMAX(N) is
begin
   if N is a leaf then
        return the estimated score of this leaf
   else
        Let N1, N2, .., Nm be the successors of N;
        if N is a Min node then
           return min{MINIMAX(N1), .., MINIMAX(Nm)}
        else
           return max{MINIMAX(N1), .., MINIMAX(Nm)}
end MINIMAX;
```

Cet algorithme s'adaptate aux jeux en considérant que la maximisation est le choix du meilleur coup possible et que la minimisation représente le choix de l'adversaire en supposant que celui-ci choisit aussi la meilleure position pour lui (d'après nous). C'est cette dernière supposition qui est très forte.

```
double ami (const Plateau& Jeu, int profondeur ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    // Attention : éviter la copie suivante
    Plateau::possibles Liste = Jeu.liste_coup();
    double eval = -INFINI;
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l );  // je joue
         eval = MAX ( eval, ennemi ( Jeu, profondeur-1 ) ) ;
         Jeu.remettre ( *l ) ;
    }
    return eval;
}

double ennemi (const Plateau& Jeu, int profondeur ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    Plateau::possibles Liste = Jeu.liste_coup();
    double eval = +INFINI;
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l ); // l'autre joue
         eval = MIN ( eval, ami ( Jeu, profondeur-1 ) ) ;
         Jeu.remettre ( *l ) ;
    }
    return eval;
}

int main(int argc, char ** argv) {
    ...
    ami ( Jeu, profondeur_max )
    ...
}
```

12.2 α-β

ALPHA-BETA est une méthode de “Branch and Cut” réduisant le nombre de noeuds explorés par la stratégie “Minimax” : pour chaque noeud, elle calcule la valeur de la position ainsi que deux valeurs, alpha et beta.

Alpha : une valeur **toujours plus petite** que la vraie évaluation. Au début du parcours Alpha vaut -INFINI pour un noeud quelconque et la valeur de l'évaluation pour une feuille. Ensuite, pour un noeud à maximiser, alpha vaut la plus grande valeur de ses successeurs et, pour un noeud à minimiser, alpha est celui du prédécesseur.

Beta : une valeur **toujours plus grande** que la vraie évaluation. Au début du parcours beta vaut INFINI pour un noeud quelconque et la valeur de l'évaluation pour une feuille. Ensuite, pour un noeud à maximiser, beta est celui du prédécesseur et, pour un noeud à minimiser, beta vaut la plus petite valeur de ses successeurs.

Ce qui est garanti :

- La valeur d'un noeud ne sera jamais plus petite que alpha et jamais plus grande que beta.
- alpha ne décroît jamais, beta ne croît jamais.
- Quand un noeud est visité en dernier, sa valeur est celle de alpha si ce noeud est à maximiser, et celle de beta sinon.

```
double ami_ab (const Plateau& Jeu, int profondeur, double A, double B ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    alpha = A; beta = B;
    Plateau::possibles Liste = Jeu.liste_coup();
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l ); // je joue
         alpha = MAX( alpha, ennemi_ab ( Jeu, profondeur-1, alpha, beta ) );
         Jeu.remettre ( *l ) ;
         if (alpha >= beta) return beta;
    }
    return alpha;
}

double ennemi_ab (const Plateau& Jeu, int profondeur, double A, double B ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    alpha = A; beta = B;
    Plateau::possibles Liste = Jeu.liste_coup();
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l ); // l'autre joue
         beta = MIN( beta, ami_ab ( Jeu, profondeur-1, alpha, beta ) );
         Jeu.remettre ( *l ) ;
         if (alpha >= beta) return alpha;
     }
    return beta;
}

int main(int argc, char ** argv) {
    ...
    ami ( Jeu, profondeur_max, -INFINI, INFINI )
    ...
}
```

13 Variantes et optimisations d'Alpha-Beta

En général, les améliorations de l'algorithme alpha–beta sont de trois types :

Trier : pour améliorer l'ordre d'examen des noeuds. Cela permet de faire beaucoup plus de coupes.

Réduire : plus la fenêtre de recherche $(\beta - \alpha)$ est petite, plus il y a de coupes.

Réutiliser : sauvegarde des résultats pour le cas où ils ré-apparaîtraient (parce qu'une position est accessible par différents mouvements, parce que l'on a relancé alpha–beta, avec des paramètres différents, etc.).

Dans la suite, le **facteur de branchement** désigne la taille moyenne de la liste des coups à chaque étape (environ 35 aux échecs, de 8 à 10 aux dames, etc.).

La **profondeur** d'un algorithme de recherche, est le nombre de demi-coups que celui-ci a exploré en avance (aux échecs les meilleurs programmes regardent actuellement 8/9 demi-coups, cela monte à 11/12 pour le jeu othello, entre 9 et 12 pour les dames internationales 10×10.

14 Méthodes de tri

14.1 Méthode de l'ordonnancement quantitatif

On explore en premier les situations dont le nombre de coups légaux issus est le plus faible.

14.2 Descente itérative

Avec un facteur de branchement proche de 8, le coût d'un alpha–beta de profondeur $d+1$ est environ de 3 à 4 fois plus élevé que le coût de l'algorithme de profondeur d. Ainsi, il ne coûte pas tellement plus cher d'exécuter l'algorithme à une profondeur d, puis à une profondeur $d+1$.

L'**alpha–beta itératif** est donc en au moins deux phases : une première passe permet d'ordonner les coups du meilleur au plus mauvais à une profondeur k, l'exécution à une profondeur $k+j$ produit alors plus de coupes que si elle avait été lancée directement.

14.3 Descente itérative et gestion du temps

Un autre avantage est pour une utilisation à temps limité : on effectue un premier alpha–beta à une profondeur faible k, puis à toutes les profondeurs suivantes, jusqu'à avoir dépassé la limite de temps (ou jusqu'à ce que l'estimation prédise un dépassement au prochain appel).

15 Tables

15.1 Tables de transposition

Il est possible d'atteindre une même position par plusieurs chemins différents. On stocke alors les positions, leur évaluation et la profondeur associée dans des tables. Retrouver un noeud déjà évalué dans les tables permet alors les optimisations suivantes :

- Si le noeud a déjà été évalué à une profondeur au moins aussi grande que celle désirée, sa valeur devient celle stockée.

- Si il a été évalué à une profondeur moins grande, cette valeur sert pour ordonner les noeuds du meilleur au moins bon.

Pour permettre une recherche rapide dans les tables, celle-ci sont en général implémentées par des tables de hachage (cf section 15.3). Il est en outre possible de stocker une table par profondeur évaluée.

15.2 Tables de refutation

Le problème des tables précédentes est leur taille importante.

Avec un shéma itératif, il est possible de stocker plutôt des tables de réfutation. C'est-à-dire que pour les mauvais coups, on garde en mémoire le chemin qui indique que c'est un mauvais coup, puis on en explore la suite à une profondeur plus grande. Si le coup reste mauvais, cela suffit pour le rejeter.

15.3 Utilisation des tables de hachage de la STL

```
struct Plateau {
   long long _pb, _pn, _rb, _rn;
   double eval();
};

struct Hasher {
#define _MUL 950706376UL
#define _MOD 2147483647UL
   size_t operator()( const Plateau& _p ) const {
       size_t tmp = (_p._pb * _MUL) % _MOD;
       tmp = ((tmp + _p._pn) * _MUL) % _MOD;
       tmp = ((tmp + _p._rb) * _MUL) % _MOD;
       return  ((tmp + _p._rn) * _MUL) % _MOD;
   }
};

typedef hash_map< Plateau, double, Hasher > Hash_t;

void Memory_ab(Plateau& Jeu, Hash_t& Table, int profondeur ) {
  ...
  Hash_t::const_iterator trouve = Table.find( Jeu );
  if (trouve != Table.end() )
      return (*trouve).second;
  else
      return Table[ Jeu ] = alpha_beta(Jeu, profondeur);
}
```

Des références sur les tables de hachage sur le net :

- http://www.sgi.com/tech/stl/AssociativeContainer.html
- http://www.sgi.com/tech/stl/HashedAssociativeContainer.html
- http://www.sgi.com/tech/stl/hash_map.html

16 Fenêtres

16.1 PVS

L'algorithme PVS - Principal Variant Search - est un exemple typique d'heuristique. Il fait l'hypothèse d'une certaine "continuité" des valeurs des successeurs d'une position; en effet, en général les successeurs d'une position ont des valeurs proches. On se sert donc de la valeur du premier successeur pour définir les bornes d'une fenêtre de recherche serrée, ce qui permettra beaucoup de coupures. Cette heuristique peut ne pas marcher: si la valeur d'un successeur tombe en dehors de la fenêtre prévue, il faut refaire le calcul.

En pratique : supposons que l'on ait à maximiser et que $\alpha = 3$, $\beta = +\infty$. On effectue le prochain appel, pour le coup à évaluer suivant, par ennemi_ab avec 3 et $3 + 1$. Si le résultat vaut 3 cela veut dire que la vraie valeur est 3 ou moins, elle ne sert donc pas. Si le résultat vaut 4, cela veut dire que la vraie valeur est 4 ou plus, il faut donc relancer ennemi_ab avec $\alpha = 4$ et le précédent β.

16.2 Limitation a priori de la fenêtre de recherche

Par exemple autour de la valeur de la profondeur précédente, ou bien encore autour de la valeur de la fonction d'évaluation appliquée à la situation à étudier, plutôt qu'autour de la valeur de la situation précédente (PVS).

16.3 Recherche par aspiration

L'alpha-beta commence avec les valeurs $-\infty$ et $+\infty$. Une connaissance du jeu (nombre de pions restants, etc.) permet de restreindre la taille de la fenêtre de départ.

Si la valeur tombe dans l'intervalle choisi, alors cet intervalle était correct. Sinon, il faut relancer l'alpha-beta.

17 MTD(f)

17.1 A. Plaat

La méthode MTD(f) [A. Plaat, http://www.cs.vu.nl/~aske/mtdf.html] combine les idées précédentes : réduire la fenêtre au maximum (de taille 1), même si il faut relancer l'alpha-beta plusieurs fois, et utiliser des tables pour stocker les valeurs déjà calculées (si plusieurs alpha-beta ré-évaluent les mêmes positions).

```
double MTDf(Plateau& Jeu, int depth, double f) {
  double g = (f>MINFTY)?f:0;
  double lowerbound = MINFTY;
  double upperbound = INFTY;
  double beta; int count = 0;
  for(; lowerbound < upperbound ; ++count)  {
    if (count > 5) {
      g = Memory_ennemi_ab(Jeu, depth, lowerbound, upperbound);
      break;
    }
    if (g == lowerbound)
      beta = g+1;
    else
      beta = g;
    g = Memory_ennemi_ab(Jeu, depth, beta - 1, beta);
    if (g < beta)
      upperbound = g;
    else
      lowerbound = g;
  }
  return g;
}
```

17.2 Variante à deux pas

```
double MTDFdouble(Plateau& Jeu, int depth, double f) {
  double g = (f>MINFTY)?f:0;
  double lowerbound = MINFTY;
  double upperbound = INFTY;
  double beta; int count = 0;
  for(  double diffbound = DINFTY;
        diffbound > 1 ;
        ++count, diffbound = upperbound-lowerbound)  {

    if (count > 4) {
      g = Memory_ennemi_ab(Jeu, depth, lowerbound, upperbound);
      break;
    }
    if (g == lowerbound)
      beta = g+1;
    else if (g == upperbound)
      beta = g-1;
    else
      beta = g;
    g = Memory_ennemi_ab(Jeu, depth, beta - 1, beta + 1);
    if (g < beta)
      upperbound = g;
    else if (g > beta)
      lowerbound = g;
    else
      break;

  }
  return g;
}
```

18 Heuristiques

18.1 Profondeur selective

Dans un cadre itératif, chaque coup peut être examiné avec une profondeur qui lui est propre, suivant différents critères : sa précédente évaluation était très rapide donc on incrémente la profondeur en peu plus, et vice-versa; les coups avec une forte probabilité d'être bons sont examinés avec une profondeur faible tandis que les autres sont examinés avec une plus grande profondeur

Cette heuristique rend, en général, l'ordinateur plus humain !

18.2 La recherche focalisée

Il s'agit ici d'un shéma itératif dans lequel à partir des derniers niveaux, seules les meilleures possibilités (80/90 %) sont explorées, les autres sont abandonnées. Attention, il s'agit d'une heuristique à manipuler avec précaution.

18.3 Accélerer la victoire

Pour terminer la partie, il est possible de donner une évaluation différente aux victoires suivant le nombre de pions restants, le nombre de coups qu'il faut pour atteindre cette position, etc.

18.4 Passe

Si une position semble suffisament bonne à une profondeur k, l'ordinateur peut voir ce qui se passerait si il ne jouait pas (il passe) à la profondeur $k+1$. L'idée est qu'en général ce choix est mauvais. Ainsi, si le résultat obtenu reste bon, la position doit être vraiment bonne, dans le cas contraire, il faut faire une recherche normale.