Introduction aux techniques de base pour l'intelligence artificielle

Par Damien Guichard

Date de publication : 23 août 2009

Cet article fait un survol rapide des techniques à la base de l'intelligence artificielle appliquée aux jeux de stratégie abstraite. On y discute aussi des modules d'ordre supérieur et de leur intérêt en matière de composabilité modulaire.

Vos remarques sur cet article **Commentez**

I - Introduction

Pour certains d'entre vous, la perspective de programmer un jour une IA pour des jeux de société/stratégie en tour par tour n'est pas totalement étrangère à votre apprentissage d'un langage de programmation. Pourtant, d'autres priorités ayant pris le dessus, vous avez différé vos réalisations dans ce domaine. Sans toutefois vous en désintéresser définitivement.
Plusieurs facteurs peuvent contribuer à expliquer la persistance de votre enthousiasme :

- programmer une IA est une activité à la fois ludique, noble, exigeante et motivante ;
- aucun logiciel de création de jeu n'offre une IA prête à l'emploi ;
- même si un logiciel de création offrait une IA prête à l'emploi vous avez (à juste titre) la conviction qu'elle ne serait pas adaptable aux mécanismes de jeu tels que vous les envisagez ;
- la progression faramineuse des capacités de calcul a davantage profité aux graphismes qu'à l'IA et vous avez le sentiment légitime que la marge de progression vers des comportements plus intelligents est encore énorme.

Cet article s'appuie sur vos acquis en algorithmie pour aborder quelques techniques élémentaires de l'IA.
J'ai volontairement limité le niveau algorithmique au strict minimum, non seulement afin de rester lisible par tous, mais aussi pour mieux introduire les modules d'ordre supérieur, une technique de modélisation très prometteuse pour l'intelligence artificielle.
À bien des égards la **généricité** sur les modules me paraît une démarche aussi naturelle que la généricité sur les types. J'espère que l'exemple développé sera suffisamment convaincant pour me faire pardonner cette excentricité.

II - Où trouver de la documentation sur l'IA ?

La question ce serait plutôt « Que puis-je trouver comme documentation qui me soit utile ? »
Intelligence Artificielle est un terme générique qui recouvre :

- un éventail de solutions pour des jeux/problèmes classiques dont la représentativité a stimulé des recherches suffisamment fructueuses pour que votre jeu/application en bénéficie largement. Moyennant un certain travail de personnalisation, toujours un peu artisanal, vous devriez pouvoir réaliser une IA convaincante qui représente un certain challenge pour un joueur humain ;
- un éventail de pistes concernant des jeux/problèmes dont la représentativité stimule encore des recherches actives. Prise isolément chaque piste est une avancée. Toutefois ces pistes peuvent s'exclure mutuellement par leurs choix de conception de sorte qu'il est impossible de constituer une solution complète et optimisée pour le jeu/problème concerné. Dans ce cas défavorable, même le meilleur compromis technique envisageable ne pourra pas opposer de résistance solide face à un joueur humain suffisamment expérimenté.

La meilleure garantie de succès c'est de rattacher votre jeu/problème à un classique connu et bien étudié, comme un jeu de pions (le jeu de Dames est l'exemple représentatif) ou un jeu de positionnement (le jeu d'Échecs est l'exemple représentatif). Dans tous les cas, tant que vous n'en maîtrisez pas tous les enjeux, mieux vaudra continuer à vous documenter plutôt que de vous lancer dans un effort d'implémentation qui pourrait avoir peu de chances d'aboutir.
Comme dans tous les projets un peu complexes, la situation la plus confortable est celle où vous n'avez plus qu'à assembler des composants indépendants déjà largement éprouvés.
La documentation sur l'IA est parfois pénible à aborder parce que pas assez spécifique à votre application.
Cette généralité a néanmoins ses avantages, elle vous aide à isoler les composants principaux de votre moteur IA.
La documentation est très majoritairement en anglais et se répartit en deux catégories :

- les tutoriels débutants, dont le principal intérêt est de défricher le terrain afin de faciliter votre progression dans ce domaine intimidant qu'est l'IA ;
- les rapports de recherche, ils sont souvent très difficiles à lire et pas forcément utiles pour votre application. Même quand ils paraissent très proches de vos préoccupations, il y a toujours une restriction ou un choix de conception qui fait que rien n'est utilisable dans votre projet. À lire pour piocher des concepts et des idées.

Dans tous les cas c'est vous qui ferez la conception, rien ne vous sera jamais mâché.

Pour faire une bonne IA il faut souvent innover, y compris sur le plan algorithmique.
Il n'y a pas de solution IA universelle, s'il y en avait une elle serait déjà intégrée dans un progiciel de création de jeu, vous n'auriez pas besoin de développer.
Typiquement il faudra passer en revue un large panel de solutions techniques afin d'assembler le cocktail qui sera le plus détonant sur votre application. Il faudra aussi apprendre à identifier le "maillon faible", le mauvais composant ou la mauvaise conception qui plombe la performance et la qualité du résultat. Et après l'avoir identifié, il faudra savoir faire le diagnostic qui conduira à une solution plus "inspirée".
Un exemple de tutoriel pour débutants (à lire absolument, je n'en connais pas de meilleur) : **Chess Programming**.
Un exemple de rapport de recherche (intéressant, mais plus avancé): **Solving large combinatorial search spaces**.

III - Par quoi commencer ?

Par lire le tutoriel **Chess Programming** (à commencer par les parties **Basics**).
Les techniques qui y sont présentées sont appliquées au jeu d'Échecs, mais en fait ce sont exactement les mêmes techniques qu'il faut appliquer pour n'importe quel jeu de société/stratégie.
Il faut commencer par lire ce tutoriel, par contre il ne faut pas commencer par programmer un jeu d'Échecs, c'est trop difficile, il y a trop de nouveaux concepts à intégrer simultanément.
En fait, à mon avis, il ne faut même pas commencer par un jeu à deux joueurs.
Pour se familiariser avec les techniques de base, il vaut mieux commencer par un solitaire, c'est-à-dire une IA qui cherche une solution pour un puzzle.

IV - Les problématiques de base

L'IA a deux ennemis :

- la largeur de jeu ;
- la profondeur de jeu.

Toutes les techniques de l'IA consistent à combattre ces deux ennemis, c'est-à-dire à contourner l'inflation de l'espace de recherche, en le réduisant artificiellement soit dans sa largeur soit dans sa profondeur.
L'espace de recherche c'est l'ensemble des positions de jeu atteignables à partir d'une position donnée.
La performance d'une IA (bonne ou mauvaise qualité de l'IA) dépendra en priorité de votre capacité à limiter le naturel expansionniste de l'espace de recherche. Quelle que soit la puissance de traitement, elle ne peut pas faire face à une explosion combinatoire. Par conséquent il faudra veiller à ce que l'ensemble des positions de jeu effectivement analysées par le programme soit toujours considérablement plus petit que l'espace de recherche. Dans la plupart des cas une analyse exhaustive est soit inimaginable soit impraticable dans le cadre d'une utilisation interactive. La compétence en IA consiste essentiellement à savoir limiter l'étendue de la recherche sans (trop) compromettre la qualité de la réponse.
Très grossièrement, on peut classer les jeux en trois catégories d'espaces de recherche :

- les jeux essentiellement profonds, c'est le cas des Dames ou des Échecs, il n'y a que quelques coups possibles (disons au plus une grosse vingtaine), mais la partie peut durer plusieurs dizaines de coups ;
- les jeux essentiellement larges, c'est le cas des **pentominoes**, la partie est très courte (quelques coups seulement), mais à chaque coup il y a plusieurs dizaines ou centaines de possibilités ;
- les jeux difficiles, c'est le cas notamment du Go, ce n'est pas que les autres jeux soient faciles, c'est seulement que ceux-ci sont particulièrement retors puisqu'ils ont la mauvaise idée d'être à la fois larges et profonds.

IV-A - Les pratiques de base

La compétence la plus élémentaire en IA c'est :

- savoir générer les coups possibles à partir d'une position de jeu ;

- savoir lutter un minimum contre la « fausse » largeur.

La « fausse » largeur est une largeur par redondance.
Il y a deux cas très courants qui mènent à un espace de recherche redondant en largeur, pour un jeu solitaire :

- vous faites le mouvement A, puis le mouvement B, ou bien vous faites le mouvement B puis le mouvement A, si dans les deux cas vous aboutissez à la même position de jeu alors vous avez inutilement dédoublé la recherche sur cette position. Par exemple dans le Rubik's Cube vous faites ¼ de tour face avant + ¼ de tour face arrière, c'est la même chose que faire ¼ de tour face arrière + ¼ de tour face avant. Vous devrez supprimer cette symétrie sans quoi votre espace de recherche va grossir inutilement ;
- vous faites le mouvement A, puis le mouvement B, mais le mouvement B annule l'effet du mouvement A et vous revenez à la position de départ. Par exemple dans le Rubik's Cube vous faites ¼ de tour face avant + ¾ de tour face avant, la face avant a fait un tour complet, vous revenez à la position initiale. Là encore vous devrez supprimer cette répétition sans quoi votre espace de recherche va grossir inutilement.

Le moyen le plus efficace de lutter contre la « fausse » largeur c'est la technique dite de la **table de transposition**.
Une **table de transposition** c'est l'ensemble des positions de jeu déjà atteintes par la recherche.
Comme c'est un ensemble, il ne peut pas contenir deux fois le même élément, c'est cela qui évite le « faux » élargissement de l'espace de recherche.
Je reviendrai plus loin sur la table de transposition.
Pour l'instant je vais parler du plateau de jeu afin de rendre plus concrète la notion d'espace de recherche.

V - La génération des coups

Essayons de modéliser un plateau de jeu.
Un certain plateau de jeu (**type** *board*) donne lieu à un certain type de coups (**type** *moves*).
Dans un langage de modules, ici en *Objective-Caml*, ça peut s'exprimer ainsi :

```
module type Board =
sig
  type board
  type moves
end
```

Qui se lira: il existe un certain type de modules, nommé *Board*, qui contient un certain type de plateau de jeu (*board*) et un certain type de coups (*moves*).
Jusque-là ça n'avance à pas grand-chose.
Ça commence à avancer si on dit en plus qu'une position de jeu (*game*) est un enregistrement qui contient :

- un champ *board* pour le plateau de jeu ;
- un champ *played* pour les coups qui ont été joués depuis la position initiale.

Et hop, on met à jour notre type de module :

```
module type Board =
sig
  type board
  type moves
  type game = {board: board; played: moves}
end
```

Ça n'est pas terminé.
Ce que l'on veut c'est construire l'espace de recherche.
Pour cela il nous faut une stratégie de jeu :

- une stratégie de jeu c'est une fonction qui effectue une action sur un jeu, une action, ça se note *unit*, un jeu c'est *game*, effectuer une action avec un jeu ça se note *game -> unit ;*

- cette action n'est pas quelconque, elle consiste en fait à augmenter la table de transposition avec les nouvelles positions de jeu générées à l'aide d'un coup supplémentaire ;
- ajouter une position de jeu c'est encore une action avec un jeu qui se note *game -> unit ;*
- par conséquent le type stratégie complet se note (*game -> unit*) -> (*game -> unit*), étant donné la capacité à ajouter une position de jeu il renvoie la capacité à ajouter toutes les positions atteignables en un coup

Ça nous donne ce type de module :

```
module type Board =
sig
  type board
  type moves
  type game = {board: board; played: moves}
  type strategy = (game -> unit) -> (game -> unit)
end
```

Un type de module ressemble à un type de classe, c'est une interface à laquelle un module d'implémentation devra se conformer pour prétendre à être reconnu comme un élément de ce type de module.
Cette conformité est structurelle, le module d'implémentation n'explicite pas le fait qu'il implémente un type de module, il se contente d'exporter une interface qui est structurellement un sous-type du type de module désiré.
On ne peut pas utiliser directement un type de module, il faut d'abord l'implémenter.

Avec ce type de module, on en a fini avec les plateaux de jeu.
On passe maintenant à la table de transposition.

VI - La table de transposition

Ce qu'on voudrait c'est qu'il existe une structure de données (un **type** *t*) qui réalise certaines opérations ensemblistes sur certains éléments (**type** *item*), et notamment :

- *empty*, la création d'un ensemble vide ;
- *singleton*, la création d'un singleton ;
- *add*, l'ajout d'un nouvel élément, cette fonction n'ajoutera rien du tout si l'élément est déjà présent ;
- *zip_intersect*, qui calcule l'intersection entre deux tables de transposition, le *zip* vient de ce que cette fonction utilise en plus une fonction zipper dont j'explique le rôle dans **un billet blog** ;
- *iter*, qui parcourt tout l'ensemble et applique la même fonction à chacun des éléments

La déclaration suivante dit exactement ceci :

```
module Mutable = struct
  module type Set =
  sig
    type t
    type item
    val empty: unit -> t
    val singleton: item -> t
    val add: item -> t -> unit
    val zip_intersect: (item -> item -> 'a) -> t -> t -> 'a list
    val iter: (item -> unit) -> t -> unit
  end
end
```

À savoir : il existe un type de modules, nommé *Set*, qui contient un type *t* d'éléments *item* avec les opérations *empty*, *singleton*, *add*, *zip_intersect* et *iter*.
C'est bien d'avoir déclaré ce type de module *Set*.
Ça serait encore mieux si on avait un module concret qui implémente ce module abstrait.

Bandes de veinards, **il y en a un tout prêt**, nommé *MakeSet*.
Quelques petites remarques au passage :

- la fonction *add* est de type *item -> t -> unit* ce qui nous suggère qu'elle se prête bien à l'implémentation d'une *strategy* quand on a l'identité de type *item = game ;*
- *MakeSet* est un module paramétré, c'est-à-dire une fonction d'un module vers un module.

On appelle modules d'ordre supérieur (*higher-order* modules) les modules qui peuvent être passés en paramètre ou retournés en résultat par un module paramétré.

Un module paramétré se déclare et s'utilise comme une fonction.
Dans son en-tête un module paramétré déclare des arguments modules formels.
Dans une application modulaire les arguments sont appliqués à un module paramétré en rattachant les paramètres modules réels à leurs arguments modules formels respectifs.
Les modules paramétrés sont curryfiés, le résultat de l'application modulaire partielle est un nouveau module paramétré par les arguments modules formels encore non rattachés à des paramètres modules réels.
Le résultat de l'application modulaire totale est un nouveau module d'implémentation.

Le type de module pris en paramètre par le module paramétré *MakeSet* est déclaré comme ceci :

```
module Ordered = struct
  module type Type =
  sig
    type t
    val  compare: t -> t -> int
  end
end
```

Il s'agit essentiellement d'une relation ordre total que *MakeSet* utilise pour implémenter efficacement certaines opérations ensemblistes comme l'appartenance et l'intersection.
Comme on a bien l'intention de stocker des positions de jeu dans la table de transposition il est tout à fait opportun de modifier le type de module *Board* pour l'obliger à sous-typer la relation d'ordre total *Ordered.Type*.
On ajoute également l'égalité *type t = game* afin de s'assurer que l'on compare bien des positions de jeu :

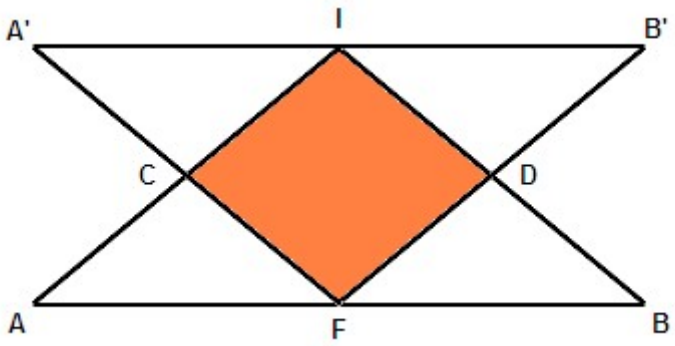
```
module type Board =
sig
  type board
  type moves
  type game = {board: board; played: moves}
  type strategy = (game -> unit) -> game -> unit
  type t = game
  val  compare: t -> t -> int
end
```

Avec cette modification, le type de module *Board* devient compatible avec le type de module *Ordered.Type*.

VII - La recherche bidirectionnelle

Maintenant ce qu'on voudrait c'est pouvoir solutionner n'importe quel type de jeu solitaire. Souvent dans ces jeux on a une position initiale et il faut trouver une séquence de coups permettant d'atteindre une position finale.
Comme, avec la table de transposition, on s'est déjà bien attaqué à limiter l'inflation en largeur, cette fois-ci on va attaquer le problème dans sa profondeur.
L'idée de la **recherche bidirectionnelle** c'est qu'on va manger la banane par les deux bouts.
On va explorer l'espace de recherche dans le sens direct, à partir de la position initiale, et on va en même temps explorer l'espace de recherche à rebours, à partir de la position finale.
Quand les deux recherches se rejoignent, on a trouvé une solution.
Chacune des deux directions de recherche possède sa propre table de transposition.

Les deux recherches se rejoignent quand l'intersection des deux tables de transposition est non vide.
Les deux recherches progressent de concert, elles vont donc se rejoindre à mi-profondeur, pour nous c'est tout bénéfice, car ça veut dire qu'on a divisé par deux la profondeur de l'espace de recherche. Une petite figure peut aider à visualiser les différentes stratégies d'exploration.



- le point I figure la position initiale et le point F figure la position finale ;
- la surface du triangle IAB figure l'espace à explorer à partir de I (recherche directe) avant d'atteindre la position F ;
- la surface du triangle FA'B' figure l'espace à explorer à partir de F (recherche à rebours) avant d'atteindre la position I ;
- la surface du losange orange ICDF figure l'espace à explorer à partir de I et F (recherche bidirectionnelle) avant la jonction quelque part sur le segment CD.

Cependant, cette figure ne rend pas complètement justice à l'efficacité de la recherche bidirectionnelle.
Pour avoir une meilleure idée de l'économie réalisée, rien ne vaut une petite simulation chiffrée :

- supposons qu'à chaque coup il n'y ait que 2 choix possibles ;
- supposons que la solution soit à une profondeur de 20 coups ;
- avec la recherche directe, il faut explorer 220 soit 1 048 576 positions ;
- avec la recherche bidirectionnelle, on trouve la solution à mi-profondeur, il faut alors explorer 210 positions pour le sens direct et autant dans l'autre sens soit en tout 2048 positions ;
- bilan de la recherche bidirectionnelle : on a exploré 29 soit 512 fois moins de positions, autrement dit on a trouvé la solution 512 fois plus vite qu'avec une recherche non ciblée.

Notre module pour solutionner les jeux de solitaire est un higher-order module qui se présente ainsi :

```
module Solver (B: Board) (Set: Mutable.Set with type item = B.game) =
struct
  let play p sa = ...
  let solve p zip ga gb = ...
end
```

En particulier il attend :

- un module qui implémente un type de plateau de jeu ;
- un module qui implémente un type table de transposition (*MakeSet* fera très bien l'affaire) dont le type d'élément sera une position de jeu (**type** *item* = *B.game*).

Et il fournit deux fonctions qu'il nous reste à implémenter :

- *play*, qui construit une table de transposition *sb* qui va un coup plus loin que la table *sa ;*
- *solve*, qui trouve tous les chemins menant de la position de jeu *ga* à la position de jeu *gb* en utilisant la stratégie du joueur *p.*

play est un petit peu alambiquée, il s'agit, pour un certain ensemble *sa* de le parcourir et d'ajouter (*Set.add*) toutes les positions de jeu générées par la stratégie de *p* à un ensemble *sb* initialement vide (*Set.empty*).

```
let play p sa =
    let sb = Set.empty () in
    Set.iter (fun g -> p (fun x -> Set.add x sb) g) sa; sb
```

À la fin *play* renvoie *sb* comme nouvelle table de transposition. Quant à *solve*, elle alterne entre *even* et *odd* pour augmenter tantôt la recherche à partir de la position initiale (table *sa*), tantôt la recherche à partir de la position finale (table *sb*).
Elle s'arrête quand l'intersection entre *sa* et *sb* est non vide.

```
  let solve p zip ga gb =
    let rec even sa sb =
      match Set.zip_intersect zip sa sb with
      | [] -> odd (play p sa) sb
      | l  -> l
    and odd sa sb =
      match Set.zip_intersect zip sa sb with
      | [] -> even sa (play p sb)
      | l  -> l
    in
    even (Set.singleton ga) (Set.singleton gb)
```

Étant donné le peu de code que l'on a écrit, on pourrait avoir du mal à penser qu'on a résolu les jeux de solitaire. Pourtant c'est largement le cas.

VII-A - Illustration avec le Rubik's Cube

Le Rubik's Cube classique serait un peu trop laborieux pour cet exemple.
Afin de simplifier les notations on va se contenter du **Pocket Cube**, la version 2 x 2 x 2 du cube classique.

Ce qu'on va faire c'est un module *PocketCube* qui va implémenter le type de modules pour nos plateaux de jeu (*Board*).
Ce sera un plateau de jeu en 3D !

```
module PocketCube =
struct
  ...
end
```

Le *PocketCube* est constitué de huit petits cubes.
On va imaginer qu'on en maintient un fixe (le supérieur avant gauche) et que seuls les autres sont mobiles autour, il en reste sept de C1 à C7.

```
  type cubie =
    | C1  | C2  | C3  | C4  | C5  | C6  | C7
```

Ils ont trois orientations possibles.

```
  type orientation =
    | O1 | O2 | O3
```

L'état complet du Pocket Cube est un septuplet de cubes et un septuplet d'orientations.

```
type 'a septuple =
  'a * 'a * 'a * 'a * 'a * 'a * 'a

type board =
  cubie septuple * orientation septuple
```

Les mouvements possibles sont le quart de tour devant/derrière/gauche/droite/dessus/dessous dans le sens des aiguilles d'une montre.
On définit aussi les types du module-type *Board*, *moves* (la liste de mouvements) et *game* (les positions de jeu).

```
type move =
  | Front  | Back  | Left  | Right  | Up  | Down
type moves =
  move list
type game =
  {board: board; played: moves}
```

L'orientation des cubes subit soit une rotation directe soit une rotation inverse.

```
let ror = function
  | O1 -> O2 | O2 -> O3 | O3 -> O1
let rol = function
  | O1 -> O3 | O2 -> O1 | O3 -> O2
```

L'état initial du Pocket Cube. Pour la résolution ce sera en fait l'état final recherché.

```
let initial = {
    board  = (C1,C2,C3,C4,C5,C6,C7),(O1,O1,O1,O1,O1,O1,O1);
    played = [] }
```

L'effet des six rotations possibles sur une position de jeu *g*.

```
let right g =
  let (c1,c2,c3,c4,c5,c6,c7),(o1,o2,o3,o4,o5,o6,o7) = g.board in {
  board  = (c2,c5,c3,c1,c4,c6,c7),(ror o2,rol o5,o3,rol o1,ror o4,o6,o7);
  played = Right::g.played }

let back g =
  let (c1,c2,c3,c4,c5,c6,c7),(o1,o2,o3,o4,o5,o6,o7) = g.board in {
  board  = (c1,c2,c3,c5,c6,c7,c4),(o1,o2,o3,ror o5,rol o6,ror o7,rol o4);
  played = Back::g.played }

let down g =
  let (c1,c2,c3,c4,c5,c6,c7),(o1,o2,o3,o4,o5,o6,o7) = g.board in {
  board  = (c1,c3,c6,c4,c2,c5,c7),(o1,o3,o6,o4,o2,o5,o7);
  played = Down::g.played }

let left g =
  let (c1,c2,c3,c4,c5,c6,c7),(o1,o2,o3,o4,o5,o6,o7) = g.board in {
  board  = (c4,c1,c3,c5,c2,c6,c7),(ror o4,rol o1,o3,rol o5,ror o2,o6,o7);
  played = Left::g.played }

let front g =
  let (c1,c2,c3,c4,c5,c6,c7),(o1,o2,o3,o4,o5,o6,o7) = g.board in {
  board  = (c1,c2,c3,c7,c4,c5,c6),(o1,o2,o3,ror o7,rol o4,ror o5,rol o6);
  played = Front::g.played }

let up g =
  let (c1,c2,c3,c4,c5,c6,c7),(o1,o2,o3,o4,o5,o6,o7) = g.board in {
  board  = (c1,c5,c2,c4,c6,c3,c7),(o1,o5,o2,o4,o6,o3,o7);
  played = Up::g.played }
```

scramble permet de mélanger une position de jeu *g* à l'aide de *n* rotations aléatoires.

```
let rec scramble g n =
  if n = 0 then g
  else
    scramble
    ( match Random.int 6 with
    | 1 -> front g | 2 -> back  g
    | 3 -> left  g | 4 -> right g
    | 5 -> up    g | n -> down  g )
    (n-1)
```

inverse permet d'inverser une rotation et *zip* permet de reconstituer une suite de rotations à partir d'une suite commençant de la position initiale *ga* et d'une suite conduisant à une position finale *gb*.

```
let inverse = function
  | Front -> Back  | Back  -> Front
  | Left  -> Right | Right -> Left
  | Up    -> Down  | Down  -> Up

let zip ga gb =
  List.rev_append (List.map inverse gb.played) ga.played
```

player est un joueur qui suit une stratégie simpliste : essayer les six rotations possibles.

```
type strategy =
  (game -> unit) -> (game -> unit)

let player : strategy = fun f g ->
  f (front g); f (back  g);
  f (left  g); f (right g);
  f (up    g); f (down  g)
```

Ce qui reste est destiné à satisfaire la contrainte imposée par le *Solver*, à savoir **type** *item* = *Board.game*.

```
type t = game

let compare : t -> t -> int =
  fun ga gb -> Pervasives.compare ga.board gb.board
```

VIII - Mise en œuvre du solutionneur de Pocket Cube

Ce que l'on a, c'est un solveur général à partir duquel on va dériver un solveur de Pocket Cube.
C'est tout le principe des modules paramétrés : le module utile s'obtient en instanciant le module paramétré.
Un module paramétré permet de fabriquer des modules utiles de la même façon qu'une classe permet de fabriquer des objets utiles.
D'abord un petit alias pour alléger la notation :

```
module Cube = PocketCube
```

Maintenant on crée le module *CubeSolver* qui est un solveur de Pocket Cube.
Pour faire un solveur de cubes à partir du solveur générique il faut lui passer *Cube* en premier paramètre et une table de transposition pour *Cube* en second paramètre.

```
module CubeSolver = Solver(Cube)(Mutable.MakeSet(Cube))
```

On appelle *application modulaire* cette façon de créer un nouveau module en fournissant des modules arguments à un module paramétré.
On initialise le générateur de nombres aléatoires.

```
Random.self_init ()
```

On crée un pocket-cube mélangé avec 60 rotations aléatoires.

```
# let scrambled = Cube.scramble Cube.initial 60;;
```

Cette ligne de commande nous renvoie ce résultat :

```
val scrambled : Cube.game =
  {Cube.board =
    ((Cube.C3, Cube.C4, Cube.C2, Cube.C7, Cube.C1, Cube.C5, Cube.C6),
     (Cube.O2, Cube.O1, Cube.O3, Cube.O1, Cube.O1, Cube.O3, Cube.O2));
   Cube.played =
    [Cube.Left; Cube.Left; Cube.Left; Cube.Up; Cube.Front; Cube.Front;
     Cube.Up; Cube.Down; Cube.Right; Cube.Right; Cube.Left; Cube.Right;
     Cube.Left; Cube.Right; Cube.Right; Cube.Up; Cube.Down; Cube.Up;
     Cube.Front; Cube.Front; Cube.Left; Cube.Back; Cube.Left; Cube.Right;
     Cube.Up; Cube.Up; Cube.Up; Cube.Back; Cube.Front; Cube.Up; Cube.Back;
     Cube.Front; Cube.Up; Cube.Down; Cube.Up; Cube.Front; Cube.Left;
     Cube.Left; Cube.Down; Cube.Front; Cube.Front; Cube.Left; Cube.Left;
     Cube.Front; Cube.Up; Cube.Left; Cube.Down; Cube.Right; Cube.Back;
     Cube.Front; Cube.Left; Cube.Up; Cube.Front; Cube.Right; Cube.Front;
     Cube.Down; Cube.Back; Cube.Left; Cube.Right; Cube.Right]}
```

Remarque : le champ *Cube.played* contient la liste des coups à jouer pour reconstituer le cube initial.
On force l'oubli du cheminement vers la position initiale.

```
# let scrambled_blanked = {scrambled with Cube.played = []};;
```

Résultat :

```
val scrambled_blanked : Cube.game =
  {Cube.board =
    ((Cube.C3, Cube.C4, Cube.C2, Cube.C7, Cube.C1, Cube.C5, Cube.C6),
     (Cube.O2, Cube.O1, Cube.O3, Cube.O1, Cube.O1, Cube.O3, Cube.O2));
   Cube.played = []}
```

La reconstitution d'une solution par recherche bidirectionnelle montre qu'il existait quatre façons d'atteindre cette configuration en 10 mouvements au lieu des 60 générés aléatoirement.

```
# CubeSolver.solve Cube.player Cube.zip scrambled_blanked Cube.initial;;
```

Résultat :

```
- : Cube.move list list =
[[Cube.Front; Cube.Up; Cube.Back; Cube.Left; Cube.Left; Cube.Down; Cube.Down;
  Cube.Right; Cube.Up; Cube.Up];
 [Cube.Up; Cube.Right; Cube.Down; Cube.Down; Cube.Back; Cube.Right;
  Cube.Front; Cube.Right; Cube.Up; Cube.Back];
 [Cube.Down; Cube.Down; Cube.Right; Cube.Back; Cube.Down; Cube.Down;
  Cube.Left; Cube.Left; Cube.Up; Cube.Back];
 [Cube.Front; Cube.Down; Cube.Right; Cube.Right; Cube.Down; Cube.Down;
  Cube.Front; Cube.Left; Cube.Up; Cube.Up]]
```

Grâce à la table de transposition, la recherche est très rapide, quelle que soit la configuration, elle ne demande guère plus d'une seconde à l'interpréteur de bytecode. Pourtant le nombre de configurations possibles est de 3 674 160, mais grâce à la recherche bidirectionnelle seule une petite fraction de cet espace de recherche est réellement explorée.

IX - Conclusion

Avec cette introduction j'ai fait un tour d'horizon des techniques pour solutionner les jeux de solitaire.
Ces techniques restent d'actualité dans le cadre d'un jeu de stratégie opposant plusieurs joueurs.

Toutefois les jeux à deux joueurs sont plus complexes, ils nécessitent des techniques supplémentaires pour lesquelles il vous faudra lire et comprendre un bon tutoriel comme **Chess Programming**.
Le jeu de Go est un jeu populaire, particulièrement en Asie, et sa largeur est telle qu'il représente encore un défi pour nombre de chercheurs en IA. Vous pourrez trouver dans l'article **MoGo, maître du jeu de Go ?** Un bilan de l'état de l'art du jeu de Go informatique.

X - Discussion sur les higher-order modules

J'espère aussi vous avoir convaincu que la modélisation avec les higher-order modules est un moyen efficace de factorisation qui vous évitera d'avoir à repasser encore et toujours sur des problèmes que vous avez déjà résolus par ailleurs. Bien sûr cette généralité n'est pas gratuite, elle a un coût intellectuel, il est plus difficile de concevoir un higher-order module que d'écrire du code qui fait le même travail pour une situation établie.
Généralement il faut arbitrer entre deux écueils possibles :

- l'*over-engineering*, qui consiste à anticiper trop tôt une complexité qui n'apparaîtra jamais parce qu'en pratique on ne rencontrera que des cas simples ;
- le bricolage, qui consiste à s'apercevoir trop tard que les solutions faites à la va-vite ne passent pas à l'échelle lorsqu'on augmente la complexité des spécifications.

Avec une IA à deux joueurs ou plus on est certain que les spécifications vont croître considérablement en complexité. Le risque est donc celui d'un bricolage qui empêcherait la réutilisation d'une boîte à outils (génération des coups, table de transposition) bien appréciable lorsqu'il faudra en plus ajouter des choses sophistiquées comme de l'*alpha-bêta*, des *heuristiques* et de l'*iterative-deepening*. Ce qu'on voudrait, c'est au contraire, pouvoir assembler des éléments pris parmi un jeu de composants paramétrables par les nouvelles conditions d'utilisation. Les higher-orders modules permettent cela avec un niveau de précision, de sécurité et de généricité impossible à atteindre avec les autres approches similaires dans leurs objectifs. Les autres approches ne font que renforcer la confiance sans renforcer la qualité technique ou bien font des compromis en choisissant de sacrifier certaines qualités techniques afin de dégager de la marge pour en améliorer d'autres.
L'avantage des higher-order modules sur la modélisation classique (*Design patterns* et/ou *UML*) c'est qu'on y capture la conception dans le langage de programmation lui-même (*Objective-Caml*) du coup :

- on est assisté pour un système de type extrêmement rigoureux et en mesure d'invalider toute incohérence dès l'étape de conception ;
- la transition vers le langage d'implantation est immédiate et sans surprise puisqu'il s'agit du même langage que le langage de conception ;
- on est certain d'éviter l'écueil de la confusion entre spécification et communication. Typiquement on peut s'imaginer qu'on a spécifié quelque chose par le simple fait qu'on a apposé une nomenclature standardisée qui fait autorité (*patterns* et/ou *UML*) sur une conception ou une ébauche de conception dont la mise en œuvre révèlera plus tard les déficiences et les insuffisances que les bons mots et les beaux diagrammes avaient masquées au moment de la validation. Avec les higher-order modules ce comportement de bluff ou d'autosuggestion n'est pas totalement écarté, mais la désillusion et la sanction viendront beaucoup plus vite et beaucoup plus fort (il n'y a pas de transtypage en *Objective-Caml*). Avec les higher-order modules on ne peut pas dessiner des belles boîtes ou faire des jolies bulles de savon et prétendre que c'est du génie logiciel ;
- contrairement aux *Design Patterns* on ne va pas à l'encontre du paradigme naturel du langage d'implantation, on ne cherche pas à faire des tours de passe-passe dans un langage de contorsionniste. Au final on n'y perd rien en sécurité (bien au contraire) et ce qu'on y perd en lisibilité n'est strictement dû qu'au niveau supérieur de spécification, il n'y a pas de programmation *ravioli* où il faudrait suivre les méandres infinis d'un fil d'exécution arbitrairement obscurci par une conception soi-disant plus flexible.

XI - Pour aller plus loin

Les modules paramétrés existent dans d'autres langages qu'*Objective Caml*.
En fait le langage de modules est largement indépendant du système de typage sous-jacent et n'importe quel langage de programmation modulaire pourrait se voir doté de modules paramétrés.

Mais pour l'instant cela reste encore un privilège des langages de la famille ML.
Toutefois des langages évolués commencent à émerger sur des plateformes comme la JVM à l'aide desquels il serait envisageable de **simuler** les **modules paramétrés**.
La liste suivante se veut un aperçu des ressources web sur les modules paramétrés dans les deux langages où ils sont le mieux établis.
Standard ML (SML) :

- **Programming in Standard ML**

(de loin le document le plus complet sur les modules d'ordre supérieur).
Objective-Caml :

- **OCaml: The module system**
- **Modules paramétrés**
- **Les modules d'O'Caml**
- **Modules et Foncteurs**
- **The OCaml module system**
- **Implementation as ML Functors**
- **Tilings as a programming exercise**