Algorithmes pour les jeux
Stratégies min-max et alpha-béta

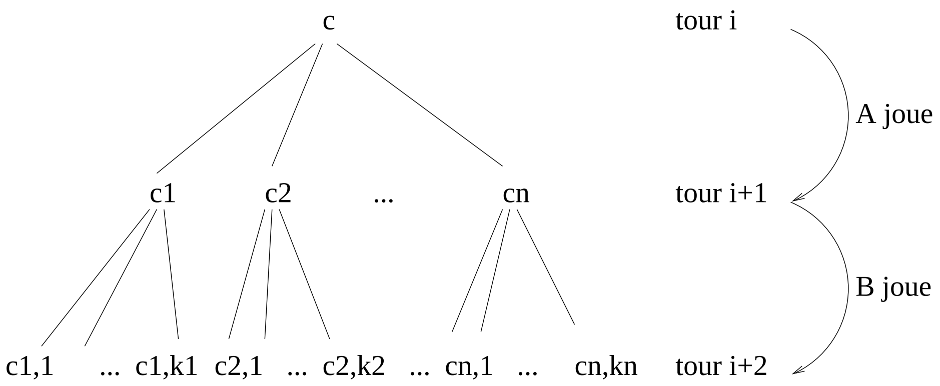
Considérons un jeu à deux joueurs, appelons-les A et B, où la configuration d'une partie est une donnée finie, et où chaque joueur n'a, à chaque tour, qu'un nombre fini de coups possibles. C'est le cas de la majorité des jeux de plateau à deux joueurs. On se propose ici de programmer un algorithme classique pour les jeux, *la stratégie min-max* puis une optimisation de celle-ci, la stratégie *alpha-béta*.

Principe

On appelle *configuration* une description complète de l'état d'une partie. Par exemple, au *puissance 4*, une configuration est une description de la grille et l'information "c'est à X de jouer". Aux échecs, une configuration est une description de l'échiquier, l'information "c'est aux blancs de jouer" (ou aux noirs), tels pions ont avancé de 2 cases initialement, mais aussi toutes les configurations précédentes (pour les répétitions de position), éventuellement le temps de jeu de chaque joueur, etc.

On suppose que l'on dispose également d'une *fonction d'évaluation*, qui associe à toute configuration une valeur réelle, avec des valeurs particulières pour la défaite, la victoire, et la partie nulle éventuellement. Comme son nom l'indique, cette fonction évalue la position d'un joueur : plus cette valeur est grande et plus le joueur est en bonne position. Il est clair que chaque joueur essaie de maximiser sa fonction d'évaluation et de minimiser celle de l'autre joueur.

La stratégie *min-max* est fondée sur cette idée. Supposons que nous sommes dans une configuration c et que c'est au joueur A de jouer. Soit f la fonction d'évaluation de A. L'arbre des coups possibles à partir de c sur un tour est représenté ci-dessous :



Si A joue le coup i, qui le mène en position c_i, alors B cherchera à minimiser la fonction d'évaluation de A au coup suivant, c'est-à-dire choisira la configuration $c_{i,j}$ qui minimise f. A a donc intérêt à choisir le coup i qui maximise ce minimum. C'est la stratégie *min-max.* A choisit donc de jouer le coup i_0 tel que

$$\min_{1\leq j\leq k_{i_o}} f(c_{i_0,j}) = \max_{1\leq i\leq n} \min_{1\leq j\leq k_i} f(c_{i,j})$$

A évalue donc les positions c_i selon la fonction $\min_{1\leq j\leq k_i} f(c_{i,j})$.

On peut itérer plus loin cette méthode, en prenant comme évaluation sur la configuration $c_{i,j}$ non pas la fonction f mais l'évaluation rendue par la méthode min-max elle-même. L'itérer n fois, c'est prévoir $2n$ coups à l'avance.

1 Préliminaires

Question 1 *Écrire une fonction* `minimize` *qui prend une liste l, une fonction f sur les éléments de l et une fonction* `comp` *de comparaison des résultats de f (*`comp a b` *est vrai si $a \leq b$) et qui renvoie l'élément qui réalise le minimun de f sur l selon* `comp`*, ainsi que ce minimun.*

`minimize comp f l` *est donc du type suivant :*

`minimize : ('a -> 'a -> bool) -> ('b -> 'a) -> 'b list -> ('b,'a)`

Question 2 *En déduire deux fonctions* `min_list` *et* `max_list` *qui prennent une fonction* `f` *de type* `'a -> float` *et une liste* `l` *de type* `'a list` *et qui rende un couple* `(x,v)` *qui minimise* `f`*.*

Remarque 1 *Les fonctions* `min_list` *et* `max_list` *réalisent l'algorithme* min-max *sur un demi-pas.*

2 Stratégie min-max

Soit `'a` le type des configurations du jeu. Soit `eval` la fonction d'évaluation, de type `'a -> float`. On suppose que cette fonction rend `0.0` pour une défaite, `1.0` pour une victoire, et un réel de $]0,1[$ sinon. On suppose également qu'il n'y a pas de partie nulle. Soit `suiv` une fonction, de type `'a -> 'a list` rendant, pour une configuration donnée, la liste des configurations suivantes possibles, c'est-à-dire la liste des coups possibles.

Question 3 *Ecrire une fonction* `minmax` *prenant en arguments* `eval`*,* `suiv` *et une configuration* `c` *et rendant le coup à jouer par la stratégie* min-max*, pour un seul niveau d'analyse (c'est-à-dire 2 coups).*

Question 4 *Ecrire une fonction* `minmax_n` *prenant les même arguments, et un entier* `n`*, et rendant le coup à jouer pour une analyse sur 2*`n` *coups.*

3 Application : le jeu des allumettes

On va appliquer la stratégie précédente au jeu bien connu des allumettes : Initialement, il y a un certain nombre d'allumettes sur la table. Chaque joueur, à son tour, prend 1, 2 ou 3 allumettes. Celui qui prend la dernière allumette a perdu. Il y a une stratégie gagnante pour ce jeu, et nous allons avoir que la méthode min-max permet de la trouver.

On choisit comme type des configurations le couple d'un booléen, qui indique quel joueur doit jouer, et une entier qui indique combien il reste d'allumettes sur la table.

```
type configuration == bool * int
```

Question 5 *Ecrire une fonction* `affiche_configuration` *affichant une configuration. Par exemple, la configuration* `(true,13)` *pourra être affichée :*

```
au joueur 1 : |||||||||||||
```

Question 6 *Ecrire une fonction* `suiv` *rendant, pour une configuration, la liste des configurations suivantes possibles.*

Question 7 *Ecrire une fonction* `eval` *d'évaluation d'une configuration (la plus simple possible, la seule chose qui compte est qu'elle indique quand la partie est gagnée).*

Question 8 *Ecrire une fonction* `joue` *qui associe à une configuration la configuration suivante, obtenue avec la stratégie min-max appliquée à une profondeur suffisante pour trouver la stratégie gagnante (si elle existe).*

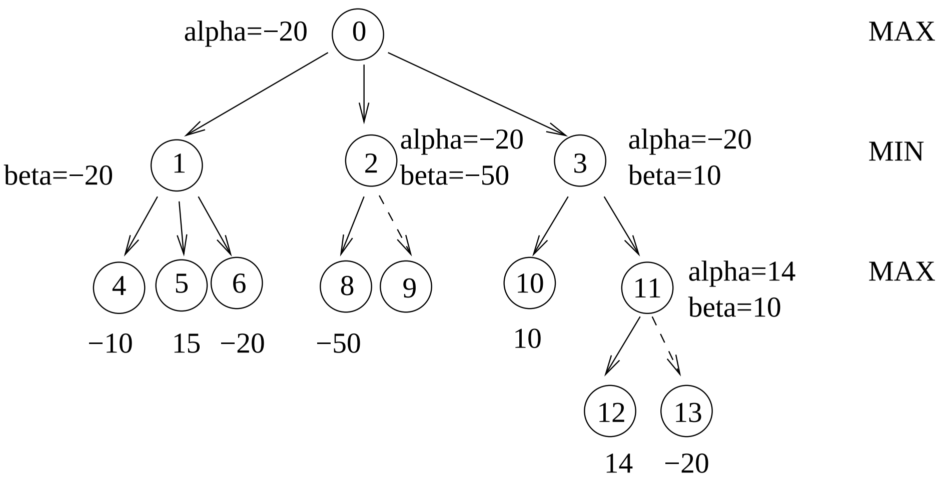
Question 9 *En déduire enfin une fonction* `jeu` *qui prend en argument les fonctions de jeu de chaque joueur (du même type que la fonction* `joue` *et qui fait jouer ces joueurs l'un contre l'autre.*

Il suffit alors de lancer `jeu` *avec la fonction* `joue` *pour les deux joueurs pour voir l'ordinateur jouer contre lui-même. Vous pouvez alors écrire facilement une fonction* `joueur` *qui demande à l'utilisateur le coup suivant (c'est à dire le nombre d'allumettes à prendre) et qui renvoie la nouvelle configuration. En donnant cette fonction et la fonction* `joue` *en argument à* `jeu`*, vous pouvez jouer contre l'ordinateur et sa stratégie* min-max*.*

4 La stratégie Alpha-Béta

L'algorithme *alpha-béta* n'est qu'une amélioration de l'algorithme *min-max*. L'algorithme *min-max* explore l'intégralité de l'arbre des coups possibles, or si l'on regarde bien, on peut éliminer certaines branches qui ne donneront pas à coup sûr une valeur supérieure de la fonction d'évalution.

Pour mieux comprendre regardons l'exemple de graphe suivant :



0 est un noeud de type MAX, c'est à dire qu'il maximise les valeurs de ses fils. 1, 2 et 3 sont des noeuds de types MIN, ils minimisent les valeurs de leurs fils, et ainsi de suite.

Après exploration complète du sous-arbre de 1, on obtient un score de -20, c'est le minimun que le deuxième joueur peut réaliser. On le place dans la variable `alpha` du noeud 0. On explore ensuite le sous-arbre 2 en espérant trouver quelque chose de plus grand que `alpha`$=-20$, or la position 8 donne -50, donc le sous-arbre 2 s'évalue forcement en dessous de -20, il ne sert donc à rien d'evaluer la position 9. On voit donc que l'on stocke dans la variable `alpha` du noeud 0 la valeur maximale et que l'on s'interdit de passer en dessous dans ces fils.

Le cas du sous-arbre 3 est complémentaire car c'est un noeud MIN : on va stocker dans une variable `beta` la valeur minimale déjà rencontrée et l'on s'interdit de passer au dessus dans ces fils. Ainsi en visitant la feuille 10, on trouve la valeur 10 que l'on place dans la variable `beta` du noeud 3. Ensuite en visitant le noeud 11, on trouve une feuille de valeur 14, supérieure à 10, donc il ne sert à rien d'évaluer la feuille 13 car le résultat en supérieur à 10. La valeur du noeud 11 est finalement 14 et celle de 3, 10. Quelle est donc la valeur finale du noeud 0 ?

Voilà l'algorithme *Alpha-Béta*. On devine qu'il va être meilleur que l'algorithme min-max car il "élague" une partie de l'arbre et diminue le nombre d'évaluation de la position.

Formalisons un peu cet algorithme. A chaque noeud de l'arbre des positions, on associe deux valeurs, `alpha` et `beta` qui suivent les règles suivantes :

- Pour un noeud MAX :
  - `alpha` vaut $-\infty$ au début puis le maximun de la valeur de ces fils au fur et à mesure qu'on les calcule
  - `beta` vaut le `beta` de son père

  - on arrête l'évaluation des fils si la valeur de l'un d'eux est supérieure à `beta`
  - la valeur finale du noeud est la valeur finale d'`alpha`
- Pour un noeud MIN :
  - `beta` vaut $+\infty$ au début puis le minimun de la valeur de ces fils au fur et à mesure qu'on les calcule
  - `alpha` vaut le `alpha` de son père
  - on arrête l'évaluation des fils si la valeur de l'un d'eux est inférieure à `alpha`
  - la valeur finale du noeud est la valeur finale de `beta`

Question 10 *Écrire une fonction* `max_list` *(resp.* `min_list`*) qui réalise l'agorithme* `alpha-béta` *sur un demi niveau d'analyse lorsque le noeud est de type MAX (resp. MIN). Elle prend comme arguments une fonction* `eval` *de type* `float -> float -> 'a -> float` *d'évaluation des positions (les deux premiers* `float` *sont l'*`alpha` *et le* `béta` *initaux de la position), l'*`alpha` *et le* `béta` *du noeud courant, de type* `float`*, une liste* `l` *de type* `'a list` *et qui rende un couple* `(x,v)` *qui maximise (resp. minimise)* `eval` *sur la liste.*

Cependant, il se peut qu'elle arrête sa recherche prématurément si elle trouve un résultat inférieur à `alpha` *(resp. supérieur à* `béta`*), dans ce cas, elle ne rend aucun état. Pour tenir compte de cette contition, on utilise le type prédéfini* `'a option` *qui se décrit ainsi :*

```
type 'a option = None | Some of 'a;;
```

Le type de retour de la fonction est donc `'a option * float` *et la fonction renvoie* `None` *si elle ne trouve rien.*

Question 11 *Écrire de la même manière que la fonction* `minmax_n` *en fonction* `alphabeta_n` *qui applique l'algorithme* alpha-béta *au rang n.*

Question 12 *Comparer la vitesse d'exécution de* `alphabeta_n` *et* `minmax_n` *pour le jeu des allumettes.*

5 Application

L'algorithme *alpha-béta* peut être appliqué avec succès sur toute sorte de jeux. Maintenant que nous avons écrit la fonction générale et polymorphe qui résoud le problème, toute la difficulté se concentre dans l'écriture de la fonction d'évaluation.

Essayez donc d'appliquer cet algorithme aux exemples suivants :
- le puissance 4 ;
- le morpion (prendre une petite grille) ;
- la reine contre les pions : sur un échiquier, les pions noirs affrontent la reine blanche. La reine gagne si elle croque tous les pions, et les pions gagnent s'ils croquent la reine ou si l'un d'eux arrive à la promotion.

Sources

- Merci à Jean-Christophe Filliâtre pour la partie du TD sur la stratégie *min-max.*
- La page `http://www.cis.temple.edu/~ingargio/cis587/readings/alpha-beta.html` pour la partie sur la stratégie *Alpha-Béta.*

Présentation

J'ai oublié de me présenter à vous depuis le premier TD, je répare cette erreur maintenant.

Je suis Loïc Le Loarer, entré à l'Ecole Polytechnique en 1997, sorti en 2000, actuellement en école d'application à l'ENST.

Les TDs et leurs corrigés seront mis en ligne sur mon site web à l'adresse : `http://www.le-loarer.org/`. Normalement, le sujet y est mis quelques jours avant le TD lui-même, vous pouvez donc y faire un tour pour voir ce qui vous attend le jeudi suivant.