T.D. I.A. : Résolution de problèmes combinatoires (partie II) correction

Pierre-Édouard Portier

1 Recherche Taboue

1.1 Solution initiale (algorithme glouton)

Algorithm 1 arbre-k glouton

aretes_libres ← toutes les arêtes de G
arbre_resultat ← ∅
while *arbre_resultat* ne possède pas k arêtes **do**
  ajouter à l'*arbre_resultat* une arête (e) des *aretes_libres* telle que

  1. e soit de poids le plus petit possible
  2. e soit connectée aux arêtes courantes de l'*arbre_resultat*
  3. e ne forme pas de cycle avec les arêtes courantes de l'*arbre_resultat*

  retirer e des *aretes_libres*
end while
return *arbre_resultat*

En appliquant l'algorithme 1 on trouve l'arbre $\{(1,2),(1,4),(4,7),(6,7)\}$ de poids 40.

1.2 Opération pour définir un voisinage

On échange une arête à l'intérieur de l'arbre-k courant par une arête à l'extérieur, le résultat devant rester un arbre. On parle d'échange *statique* quand les nœuds de l'arbre solution restent inchangés (sinon, on parle d'échange *dynamique*).

1.3 critères d'appartenance à la liste taboue

D'abord, précisons que :
– dire d'une arête ajoutée qu'elle est taboue, c'est dire qu'il est interdit de la retirer
– dire d'une arête retirée qu'elle est taboue, c'est dire qu'il est interdit de l'ajouter

En général (c'est-à-dire pour des instances intéressantes (entendre : "difficiles") du problème de l'abre-k de poids minimal), il y a plus d'arêtes à l'extérieur de l'arbre-k qu'à l'intérieur. Ainsi, une arête retirée peut rester plus longtemps taboue qu'une arête ajoutée.

Mais, existe-t-il une limite à la durée pendant laquelle une arête ajoutée peut rester taboue ? Oui, car si une arête ajoutée reste taboue pendant k itérations, après k coups, tous les coups sont tabous !

1.4 jouons quelques coups

coup	liste taboue		ajout	retrait	w
	1	2			
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	40
1	$\emptyset$	$\emptyset$	(4,6)	(4,7)	47
2	(4,6)	(4,7)	(6,8)	(6,7)	57
3	(6,8), (4,7)	(6,7)	(8,9)	(1,2)	63
4	(6,7), (8,9)	(1,2)	(4,7)	(1,4)	46
5	(1,2), (4,7)	(1,4)	(6,7)	(4,6)	37
6	(1,4), (6,7)	(4,6)	(6,9)	(6,8)	37
7	(4,6), (6,9)	(6,8)	(8,10)	(4,7)	38

1.5 critère d'aspiration

À la place du coup 2, on ne peut pas jouer *+(4,7)-(6,7)* qui donnerait pourtant une solution de poids $49 < 57$. Mais ce n'est pas grave car $49 > 40$ (40 étant le poids de la meilleure solution courante au temps du deuxième coup). Remarquons que lorsqu'une solution voisine meilleure que la meilleure solution courante n'est pas accessible à cause des listes taboues, un critère dit d'*aspiration* est souvent appliqué qui consiste à faire fi de l'interdiction imposée par les listes taboues et à rejoindre quand même la solution.

1.6 utilisation d'une mémoire à plus long terme

On a décidé arbitrairement de s'arrêter après 7 coups. Ce type de décision arbitraire est le plus souvent nécessaire car avec une méthode heuristique telle la recherche taboue on ne peut pas savoir quand un optimum global est atteint...

Ainsi, il s'agit de déterminer comment relancer intelligemment une recherche. On peut par exemple se demander quelles sont les solutions depuis lesquelles on *ne veut pas* relancer la recherche ? :

- les solutions initiales des précédentes recherches ?
- toutes les solutions par lesquelles on est déjà passé ? (possible mais en pratique pas toujours très efficace...)
- certaines solutions particulières (par exemple les *optimums locaux*) ?

Sur notre précédente recherche, ce dernier critère interdirait de partir des solutions 0, 5 et 6.

Essayons d'utiliser ce dernier critère en rendant taboues l'union des arêtes des solutions des coups 0, 5 et 6 (c'est l'ensemble $\{(1,4),(4,7),(6,7),(6,8),(8,9),(6,9)\}$) lors de la construction gloutonne d'une nouvelle solution initiale. En fait, proposons de ne s'interdire de piocher parmi ces arêtes que pour les deux premières étapes de la construction gloutonne d'une solution. Cette paramétrisation qui semble faire un peu "recette de cuisine" est souvent une étape essentielle de l'utilisation de méthodes heuristiques...!

On obtient la nouvelle solution initiale $\{(3,5),(5,9),(8,9),(8,10)\}$ qui est de poids 38 et à partir de laquelle on relance une recherche taboue :

coup	liste taboue		ajout	retrait	w
	1	2			
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	38
1	$\emptyset$	$\emptyset$	(9,12)	(3,5)	41
2	(9,12)	(3,5)	(10,11)	(5,9)	34
3	(3,5), (10,11)	(5,9)	(6,8)	(9,12)	41

En fait, la solution de poids 34 se trouve être l'optimum global...

2 optimisation $\alpha - \beta$

On représente le déroulement d'un jeu à deux joueurs sous la forme d'un arbre. Chaque nœud représente une position du jeu. Les fils d'un nœud sont les positions atteignables en jouant un coup à partir de la position de ce nœud. On décide arbitrairement d'une profondeur (MAX_DEPTH) pour l'arbre. Les joueurs ($J1$ et $J2$) jouent chacun leur tour. On se place du point de vue de $J1$. On possède une fonction d'évaluation ($eval$) qui, à une position, associe une valeur d'autant plus élevée que la position est favorable à $J1$. $J1$ cherche à atteindre une position qui maximise $eval$ tandis que $J2$ cherche à atteindre une position qui minimise $eval$.

Ainsi, en appliquant l'algorithme 2 à la position a de l'exercice, on trouve que le meilleur coup à jouer est celui qui de a fait passer à i. Remarquons qu'il a fallu explorer tout l'arbre.

Algorithm 2 $max(P, depth)$

```
if depth > MAX_DEPTH then
  return P
else
  best_val ← 0
  for all son tel que son soit un fils de P do
    son_val ← eval(min(son, depth + 1))
    if son_val > best_val then
      best_son ← son
      best_val ← son_val
    end if
  end for
  return best_son
end if
```

L'optimisation $\alpha - \beta$ améliore l'algorithme $min - max$ en déterminant des sous-arbres qu'il est inutile d'explorer. L'idée est de considérer que lorsque le programme max appelle le programme min (et réciproquement) il lui lance un défi du type : "pour l'instant la meilleure position atteignable possède une valeur β, donc si tu me proposes moins je ne considérerais pas ta proposition." Connaissant β, le programme min, dès qu'il découvre qu'un de ses fils vaut moins que β, n'évalue pas ses fils restant car max préférera dans tous les cas la position qui offre une valeur β...

Ainsi, on peut modifier les programmes max et min qui deviennent ceux des algorithmes 4 et 5.

En appliquant cette optimisation sur l'exemple de l'exercice (Figure 1), on élague les branches ($e \rightsquigarrow 8$), ($f \rightsquigarrow h$), ($k \rightsquigarrow 6$), ($n \rightsquigarrow 3$), ($o \rightsquigarrow 9$) et ($m \rightsquigarrow q$).

Algorithm 3 $min(P, depth)$

if $depth > MAX\_DEPTH$ **then**
  return P
else
  $best\_val \leftarrow \infty$
  for all son tel que son soit un fils de P **do**
    $son\_val \leftarrow eval(max(son, depth + 1))$
    if $son\_val < best\_val$ **then**
      $best\_son \leftarrow son$
      $best\_val \leftarrow son\_val$
    end if
  end for
  return $best\_son$
end if

Algorithm 4 $max(P, depth, \beta, \alpha)$

if $depth > MAX\_DEPTH$ **then**
  return P
else
  $sons\_best\_val \leftarrow 0$
  for all son tel que son soit un fils de P **do**
    if $sons\_best\_val > \beta$ **then**
      $\beta \leftarrow sons\_best\_val$
    end if
    $son\_val \leftarrow eval(min(son, depth + 1, \beta, \alpha))$
    if $son\_val > \alpha$ **then**
      return son {élagage}
    end if
    if $son\_val > sons\_best\_val$ **then**
      $best\_son \leftarrow son$
      $sons\_best\_val \leftarrow son\_val$
    end if
  end for
  return $best\_son$
end if

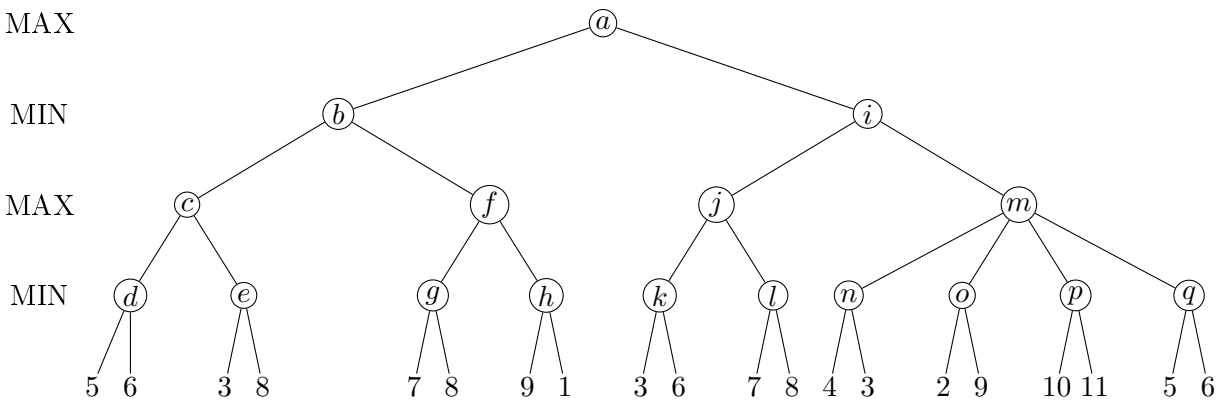

FIGURE 1 – Application de l'optimisation Alpha-Beta pour l'algorithme minimax

Algorithm 5 $min(P, depth, \beta, \alpha)$

```
if depth > MAX_DEPTH then
  return P
else
  sons_best_val ← ∞
  for all son tel que son soit un fils de P do
    if sons_best_val < α then
      α ← sons_best_val
    end if
    son_val ← eval(max(son, depth + 1, β, α))
    if son_val < β then
      return son {élagage}
    end if
    if son_val < sons_best_val then
      best_son ← son
      sons_best_val ← son_val
    end if
  end for
  return best_son
end if
```

3 21 takeaway

C'est un exemple simple de jeu de type Nim. C'est-à-dire qu'il existe une fonction d'évaluation f telle que :

- pour une position perdante P, $f(P) = 0$
- pour toute position G telle que $f(G) \neq 0$, il existe toujours un coup qui fait passer à une position P pour laquelle $f(P) = 0$
  Ici, E désignant l'ensemble des objets sur le plateau, f peut être définie comme $|E| mod 4$ (le reste de la division du nombre d'objets de E par 4).