Chapitre 7

De nouvelles heuristiques de recherche appliquées à la résolution d'Atarigo

7.1. Introduction

Atarigo est un jeu à deux joueurs et à information complète qui se joue sur des gobans de taille variable. C'est une simplification du jeu de Go. Il est souvent utilisé à ce titre pour apprendre les rudiments du jeu de Go aux enfants et aux débutants. Il se joue sur une grille carrée, et chacun son tour les joueurs posent une pierre de leur couleur sur une intersection. Les pierres sont soit noires, soit blanches, et c'est noir qui commence à jouer.

Le but du jeu est de capturer une chaîne de l'adversaire. Une chaîne est un ensemble de pierres de la même couleur reliées entre elles par les lignes entre les intersections de la grille. Par exemple les pierres blanches marquées d'un carré dans la figure 7.1 forment une chaîne de trois pierres. Une propriété importante d'une chaîne est son nombre de libertés. Une liberté est une intersection vide voisine horizontalement ou verticalement de la chaîne. La chaîne blanche de la figure 7.1 a une seule liberté qui est marquée par un rond. Un autre exemple de calcul du nombre de libertés est celui de la chaîne noire marquée par des triangles. Cette chaîne noire a quatre libertés. Une chaîne est capturée si elle n'a plus de libertés. Par exemple, si noir joue une pierre sur l'intersection vide marquée d'un rond de la figure 7.1, il capture la chaîne blanche et gagne la partie.

Chapitre rédigé par Frédéric BOISSAC et Tristan CAZENAVE.

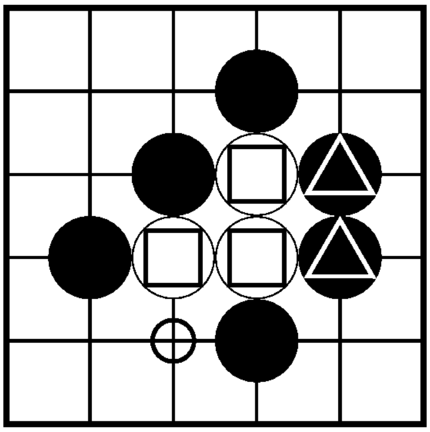

Figure 7.1. *La chaîne blanche est en atari*

Nous nous intéressons principalement dans ce chapitre aux heuristiques qui contribuent à accélérer une recherche *alpha-beta*. Les heuristiques présentées ont été implémentées par Frédéric Boissac avec le logiciel Dariush [BOI 06].

L'algorithme présenté est le premier à résoudre Atarigo 6x6 pour un goban vide et pour un goban 7x7 avec un Cross-Cut.

Dans ce chapitre la couleur pour laquelle nous chercherons une solution gagnante sera appelée couleurordi et l'autre couleurjoueur.

7.2. État de l'art

La première résolution d'Atarigo 6x6 avec quatre pierres croisées au centre (ou Cross-Cut) a été réalisée par Tristan Cazenave en 2001 [CAZ 02c]. La position initiale de ce jeu est donnée dans la figure 7.2. L'algorithme qui l'a résolu détecte les menaces pour couleurordi quand c'est à couleurjoueur de jouer. La détection des menaces permet de n'envisager qu'un petit nombre de coups pour couleurjoueur. De plus, si aucune menace n'est détectée, la branche est coupée et aucun coup n'est essayé pour couleurjoueur.

L'ajout d'heuristiques classiques comme les tables de transposition, les coups qui tuent et l'heuristique de l'historique a permis d'accélérer la résolution [CAZ 02b]. En généralisant la définition de menaces et en regroupant des ensembles de menaces similaires, on arrive à l'algorithme des menaces généralisées qui résout Atarigo 6x6

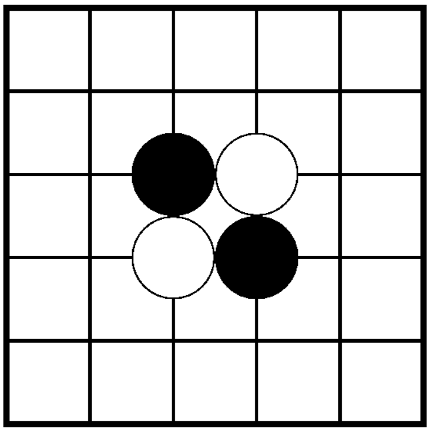

Figure 7.2.

avec quatre pierres croisées encore plus vite [CAZ 03]. On peut noter que la vérification des menaces est accélérée par l'utilisation d'heuristiques admissibles qui permettent de n'envisager qu'un sous ensemble des coups possibles et de couper le nœud lorsque la vérification de la menace est détectée comme impossible [CAZ 02a] (c'est par exemple le cas quand l'algorithme détecte qu'il est impossible pour couleurordi de prendre une chaîne de trois degrés de libertés en seulement deux coups)

Erik van der Werf a lui aussi écrit un programme de résolution d'Atarigo 6x6 avec quatre pierres croisées en 2002 [WER 02]. Son programme utilise un *alpha-beta* avec les optimisations classiques et résout le jeu plus lentement que l'approche avec les menaces. Son programme a aussi été appliqué à l'Atarigo 6x6 commençant avec un damier vide mais ne l'a pas résolu. Il a en revanche résolu Atarigo 6x6 avec une ouverture stable après quatre coups [WER 05].

7.3. Algorithme de recherche

Dans cette section nous présentons l'algorithme de recherche et les optimisations qui lui sont associées.

7.3.1. alpha-beta *principal*

L'algorithme de plus haut niveau à partir duquel sont appelées les heuristiques est un *alpha-beta* classique. Il utilise une fonction d'évaluation simple qui renvoie un si

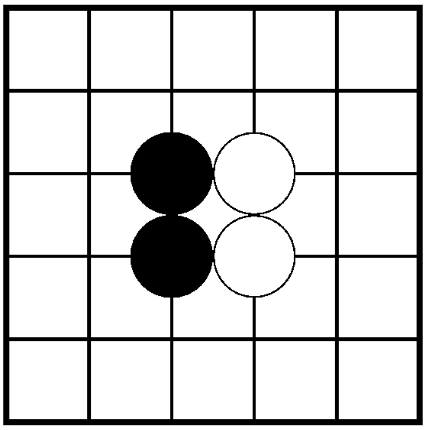

Figure 7.3.

couleurordi a gagné la partie et zéro sinon. Le but de cet algorithme est de répondre à la question suivante : "existe-t-il un coup gagnant pour couleurordi et si oui lequel ?"

Une table de transposition est associée à l'*alpha-beta*. Elle permet de détecter très rapidement qu'une position a déjà été étudiée précédemment et ainsi de ne pas la réétudier. Une table de transposition est un tableau indexé par la clé de la position. A chaque intersection de la grille est associé un nombre aléatoire pour noir et un nombre aléatoire pour blanc. La clé d'une position est calculée très rapidement par des XOR entre les nombres aléatoires correspondants aux couleurs et aux intersections des pierres présentes sur la grille. Pour une position, et donc une clé donnée, on sait en un accès mémoire dans le tableau si la position a déjà été rencontrée.

7.3.2. *Premier tri des coups possibles*

Les heuristiques utilisées pour accélérer l'*alpha-beta* sont essentiellement basées sur le tri des coups à essayer. L'algorithme utilise plusieurs méthodes de tri successives. Le premier tri consiste à privilégier les mises à un degré de liberté des chaînes adverses, puis les mises à 2 degrés, puis à 3, etc. Ensuite pour chaque sorte de coups (mise à 1, mise à 2, etc.), l'algorithme trie à nouveau pour que ces coups soient le plus proches possible d'une chaîne de la même couleur. L'idée est que globalement pour prendre un pion adverse, il faudra diminuer les degrés de liberté de l'adversaire tout en jouant des coups suffisamment forts pour ne pas se faire prendre soi-même.

7.3.3. alpha-beta *secondaire*

A l'intérieur de l'*alpha-beta* principal, un autre algorithme *alpha-beta*, plus rapide, est appelé. Son rôle est de déterminer si on peut prendre un pion adverse en x coups. Il est beaucoup plus rapide que l'*alpha-beta* principal car il est pas obligé d'étudier tous les coups d'attaque. Si par exemple on le lance pour $x = 3$ cela impose de n'étudier que les mises à 1 degré des chaînes adverses. En effet une mise à 1 degré au premier nœud de cet *alpha-beta* entraîne au mieux une prise au troisième nœud. Pour $x = 5$, on ne doit étudier que les coups de mise à 1 degré et de mise à 2 degrés, et ainsi de suite pour les x suivants. Ces optimisations sont similaires à l'utilisation d'heuristiques admissibles [CAZ 02a]. Par contre pour qu'il y ait une certitude sur le résultat d'une prise, l'algorithme étudie tous les coups possibles de défense. L'*alpha-beta* secondaire est lancé pour chaque coup de l'*alpha-beta* principal. S'il renvoie que le coup est gagnant, la position n'est pas cherchée plus avant. L'utilisation de l'*alpha-beta* secondaire est la cause principale de la vitesse de l'algorithme.

La profondeur de l'*alpha-beta* secondaire varie en fonction de la profondeur de l'*alpha-beta* principal. Elle diminue lorsque la profondeur de l'*alpha-beta* principal augmente. Une valeur de x trop grande à une profondeur très grande prendrait énormément de temps. Et bien que le nombre d'élagages de l'*alpha-beta* principal soit plus grand, le temps global d'étude augmenterait considérablement. Cette optimisation est proche de l'élargissement généralisé qui consiste à élargir progressivement la taille des menaces en partant de la racine [CAZ 04].

7.3.4. *Second tri des coups à envisager*

L'algorithme passe en revue les coups possibles dans l'ordre du premier tri. Il teste pour chacun d'eux si le coup est gagnant si l'adversaire ne répond pas, ce qui correspond à détecter certaines menaces de gains [CAZ 02c]. L'*alpha-beta* secondaire est utilisé pour détecter ces menaces. En revanche, contrairement aux menaces généralisées, l'algorithme ne coupe pas le coup si ce n'est pas une menace, mais le relègue simplement après les coups de menaces.

Le second tri consiste donc à tester les coups de menaces dans l'ordre du premier tri, puis à n'envisager que par la suite les autres coups. Il permet de gagner énormément de vitesse.

7.3.5. *Envisager en priorité les coups gagnants de l'adversaire*

Lorsqu'un coup n'amène pas à un élagage à une profondeur p, l'algorithme regarde le coup mémorisé adverse qui a donné un élagage à la profondeur $p+1$. Si ce coup n'a pas encore été étudié à la profondeur p, l'ordre des coups est modifié pour l'étudier en priorité. Cette heuristique est basée sur le principe qu'un bon coup "de réponse" pour l'adversaire a des fortes chances d 'être aussi un bon coup "d'attaque" pour la couleur étudiée (ce qui n'est malheureusement pas toujours vrai...).

Cette heuristique est similaire à celle qui a été décrite dans le cadre de la recherche heuristique sur vie et la mort de groupes au jeu de Go par Thomas Wolf [WOL 00].

7.3.6. *Les coups qui tuent*

Un coup qui a déjà donné un élagage à une certaine profondeur a des chances d'en donner à nouveau lorsque l'algorithme revient à cette profondeur. L'heuristique qui essaie en premier les coups qui ont déjà donnés un élagage à la même profondeur est appelée heuristique des coups qui tuent [MAR 86]. L'expérience a montré que l'utilité de cette heuristique peut vraiment varier suivant la situation :

- si le plateau étudié est susceptible d'avoir beaucoup de coups forcés (mise à 1 degré, coup qui tue, etc.) cette supposition est bonne. C'est par exemple le cas pour une étude qui commence avec un Cross-Cut. Cette figure de départ favorise énormément les coups forcés. Dans ce cas l'algorithme va étudier en priorité ces fameux coups qui ont déjà donné un élagage.

- par contre si la configuration du jeu n'admet pas de coups forcés à courte échéance, il peut y avoir explosion combinatoire des coups et donc une explosion combinatoire des « situations ». Dans ce cas, un coup qui a déjà donné un élagage à cette profondeur n'a peut être plus aucun rapport avec l'étude de la situation présente. Dans ce cas, étudier ce coup en priorité est une mauvaise option et il vaut mieux se contenter du premier tri. C'est par exemple le cas d'une recherche de solution gagnante sur un plateau vide.

7.3.7. *La limitation du nombre de coups envisagés*

Le second tri fait que globalement si il existe une solution gagnante pour une couleur, il y a de fortes chances que cette solution soit étudiée dans les premiers coups. Et statistiquement cela ce vérifie à chaque étage de l'*alpha-beta* principal. Partant de ce principe, le nombre de coups de couleurordi est limité à n. Ce nombre n dépend essentiellement du goban de départ. Pour un goban vide en 6x6, si $n = 9$ le rendement est optimisé. Avec un Cross-Cut en 6x6 l'optimisation est maximum pour $n = 3$. Avec un Cross-Cut en 7x7 l'optimisation est maximum pour $n = 2$.

Dans l'*alpha-beta* principal si couleurordi n'a trouvé aucune solution gagnante au bout de n coups, l'algorithme élague. Pour que le résultat final soit obtenu sans aucune incertitude, l'algorithme n'utilise cette limite que pour les coups de couleurordi, dans tous les cas couleurjoueur étudie tous ses coups. L'incertitude n'est donc que pour les résultats positifs. Si l'algorithme ne trouve pas de solution gagnante cela ne veut pas dire qu'il n'y en a pas. Par contre si il trouve une solution gagnante, celle-ci est sans aucune incertitude.

La pertinence de l'heuristique de limitation du nombre de coups envisagés est très étroitement liée au tri fait en aval.

7.3.8. *Pseudo-code de l'algorithme*

Nous donnons ci-dessous le code de l'algorithme de résolution d'Atarigo avec toutes ses heuristiques. P désigne la profondeur, qui est le nombre de coups joués depuis la position à la racine de l'arbre de recherche.

```
Variables Globales
  // Coup qui fait élaguer à profondeur
  CoupElagage : Tableau[1..PROFMAX] de Coups
  CoupOrdiQuiGagne : Coups
  // limite du nombre de coups possibles
  // pour Atarigo 6x6 Cross-Cut
  N = 2

Function Recursive(P, Jeu, Couleur, CouleurEnnemi, Resultat)

  ChercheTousLesCoups(LesCoups, CoupForcé, CoupGagnant)
  SI LesCoups.Nombre=0 ALORS
    SI Couleur=Ordi ALORS
      Resultat = 0
    SINON
      Resultat = 1
    Exit
  EtapeA
  // premier tri
  TrieLesCoups(LesCoups)
  // table de transposition
  EtudieTableTransposition(LesCoups, CoupGagnant)
  EtapeA
  SI (NON CoupForcé) ET (P<PROFMAX-10)  ET
    // détection rapide des positions gagnantes
    GagneEnXCoups(Jeu,7) ALORS
      EtapeA
```

```
  // coups qui tuent
  SI CoupElagage[P]<>0 ALORS
    MettreCoupEnPositionI(1,LesCoups,CoupElagage[P])
  EtapeB
  SI IlYADesCoupsAREvoir ALORS
    EtapeB


Function CalculeResultat(Couleur,Ordi)
  SI Couleur=Ordi ALORS
    Retourner 1
  SINON
    Retourner 0

EtapeA
  SI  CoupGagnant ALORS
    Resultat = CalculeResultat
    Exit (de la procedure Recursive)


EtapeB
  POUR I=1 A MAXCOUPS FAIRE
    JoueCoup(Jeu,LesCoups[I])
    Incremente( LesCoups[I].NombreDEtude)
    SI DegréDeLiberté(LesCoups[I])=1 ALORS
      Continue
    SI (NON CoupForcé) ET
       (LesCoups[I].NombreDEtude=1) ET
       (P<PROFMAX-14) ET
       // détection rapide des menaces
       (NON GagneEnXCoups(Jeu,5)) ALORS
         LesCoups[I].ARevoir=VRAI
         Continue
    CalculeCle(Jeu,Clé)
    Incremente(NombreDeCoupsJoués)
    Recursive(P+1,Jeu,CouleurEnnemi,Couleur,Resultat)
    MetAJourTableTransposition(Clé,Resultat)
    DeJoueCoup(Jeu,LesCoups[I])
    SI Couleur=Ordi ALORS
      SI Resultat=1 ALORS
        SI P=1 ALORS
           CoupOrdiQuiGagne=LesCoups[I]
           Exit (de la procedure Recursive)
```

```
      CoupElagage[P]=LesCoups[I]
      Exit (de la procedure Recursive)
    SINON
      // limitation du nombre de coups possibles
      SI  NombreDeCoupsJoués> N ALORS
        Resultat=0
        Exit (de la procedure Recursive)
  SINON
    SI Resultat=0 ALORS
      CoupElagage[P]=LesCoups[I]
      Exit (de la procedure Recursive)
  // A partir d'ici, il n'y a pas eu d'élagage
  // pour Couleur, CouleurEnnemi en a eu un à P+1
  // priorité aux coups gagnants de l'adversaire
  SI CoupPasEncoreEtudié(CoupElagage[P+1]) ALORS
    MettreCoupEnPositionI(I+1,LesCoups,CoupElagage[P])
```

7.4. Résultats expérimentaux

Les tests ont été effectués sur un ordinateur AMD Athlon(tm) 64 processor 3 000+ 1,79 Ghz 1Go de RAM.

Pour évaluer la vitesse de l'algorithme, on mesure le temps qui dépend de l'algorithme mais aussi de la vitesse du microprocesseur, et le nombre d 'appel de la procédure "coupjoue". Cela correspond au nombre de coups que l'algorithme a du jouer pour résoudre le problème. Ce nombre est indépendant de la vitesse du microprocesseur. Nous l'appellerons Xcoups.

- L'algorithme résout Atarigo sur un goban 6x6 avec comme figure de départ un Cross-Cut. Il résout ce problème en 141 millisecondes avec $Xcoups =$ 186 881.

- L'algorithme résout Atarigo sur un goban 6x6 vide. Il trouve le premier coup gagnant en 4 432 secondes , soit environ 1 heure 14 minutes avec $Xcoups =$ 1 354 240 513.

- L'algorithme résout Atarigo sur un goban 7x7 en démarrant la partie avec un Cross-Cut. Il y a deux sortes de Cross-Cut en 7x7 : le haut (voir figure 7.4), et le bas (cf figure 7.5). Le Cross-Cut haut est résolu en 109 millisecondes pour $Xcoups =$ 28 635. Le Cross-Cut bas est résolu en 297 millisecondes pour $Xcoups =$ 80 938.

Dans Dariush3D [BOI 06] les résultats des études décrites précédemment sont mis dans une base de données. Si Dariush3D joue avec les noirs au niveau le plus élevé

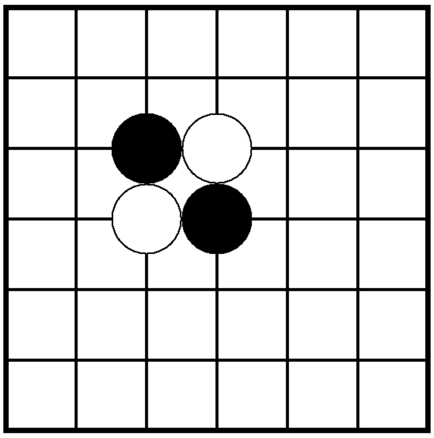

Figure 7.4. *Cross-Cut haut sur goban 7x7*

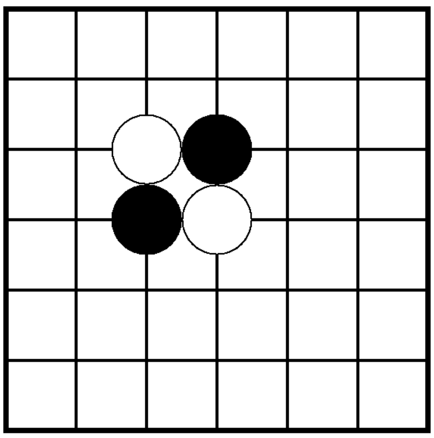

Figure 7.5. *Cross-Cut bas sur goban 7x7*

(« bon joueur »), il jouera donc forcément le coup gagnant pour un goban de taille inférieure à 6 et pour un goban 7x7 avec un Cross-Cut au départ.

Nous avons associé un acronyme à chaque heuristique :

– tt pour les tables de transposition ;

– tri pour le premier tri des coups possibles ;

– $ab1$ pour l'utilisation de l'*alpha-beta* secondaire pour détecter rapidement des positions gagnantes ;

– $ab2$ pour l'utilisation de l'*alpha-beta* secondaire pour détecter les menaces ;

– cqt pour les coups qui tuent ;

– mra pour la meilleur réponse adverse (meilleur coup à la profondeur plus un) ;

– $ncoups$ pour la limitation du nombre de coups de couleurordi.

L'évaluation des heuristiques autres que $ncoups$ dépend beaucoup de la valeur de n. Pour bien les évaluer, nous avons oté l'heuristique $ncoups$, sinon le fait de limiter le nombre de coups de couleurordi (à 2 par exemple) ne permet pas d'évaluer correctement leurs impacts).

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x		430252	1297
$sans\ tt$		x	x	x	x	x		1155436	2579
$sans\ tri$	x		x	x	x	x		1299868	3812
$sans\ ab1$	x	x		x	x	x		15590625	71250
$sans\ ab2$	x	x	x		x	x		1007735	3109
$sans\ cqt$	x	x	x	x		x		602892	1766
$sans\ mra$	x	x	x	x	x			788366	2266

Tableau 7.1. *Cross-Cut 6x6 avec l'heuristique ncoups désactivée*

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x	x	171174	437
$sans\ tt$		x	x	x	x	x	x	797188	1765
$sans\ tri$	x		x	x	x	x	x	non	non
$sans\ ab1$	x	x		x	x	x	x	4507348	20828
$sans\ ab2$	x	x	x		x	x	x	487376	1484
$sans\ cqt$	x	x	x	x		x	x	110456	281
$sans\ mra$	x	x	x	x	x		x	249304	657

Tableau 7.2. *Cross-Cut 6x6 avec* $n = 5$

Le tableau 7.1 donne Xcoups et les temps de résolution lorsqu'on n'utilise pas l'heuristique $ncoups$, c'est à dire lorsqu'on considère tous les coups possibles. Pour chaque heuristique on teste la résolution avec l'heuristique désactivée. On voit ainsi que l'heuristique qui donne les meilleurs résultats est $ab1$.

Le tableau 7.2 donne Xcoups et le temps utilisé pour $n = 5$ avec toutes les heuristiques activées (437 ms), puis avec chaque heuristique enlevée. On peut ainsi estimer l'utilité de chaque heuristique. On voit ainsi que tri et $ab1$ sont les heuristiques qui amènent le plus de rapidité. On peut aussi noter que les coups qui tuent ralentissent l'algorithme et qu'il obtient de meilleurs résultats sans les utiliser.

Les non donnés comme résultats de l'heuristique tri signifient que l'algorithme ne trouve pas de solution sans cette heuristique.

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x	x	52806	141
$sans\ tt$		x	x	x	x	x	x	67519	172
$sans\ tri$	x		x	x	x	x	x	non	non
$sans\ ab1$	x	x		x	x	x	x	2440699	11125
$sans\ ab2$	x	x	x		x	x	x	137162	375
$sans\ cqt$	x	x	x	x		x	x	65929	187
$sans\ mra$	x	x	x	x	x		x	non	non

Tableau 7.3. *Cross-Cut 6x6 avec* $n = 3$

Les meilleurs résultats pour le Cross-Cut 6x6 sont obtenus avec $n = 3$ et sont donnés dans le tableau 7.3. Le problème est résolu en 52 806 coups et 141 millisecondes. On voit que les heuristiques tri et mra sont nécéssaires à sa résolution, et que $ab1$ a un facteur d'accélération très important.

Pour $n = 2$, l'algorithme ne trouve pas de solution au Cross-Cut 6x6.

Le tableau 7.4 donne les résultats des tentatives de résolution du Cross-Cut 7x7 bas avec $n = 5$. Lorsque le temps de résolution est trop long, il n'y a pas de résultat, et celui ci est noté $>>$. On voit que les heuristiques cqt et mra sont néfastes à la résolution puisque le problème est résolu plus rapidement sans ces heuristiques.

Le tableau 7.5 donne les résultats de la résolution du Cross-Cut 7x7 bas avec $n = 2$. Le meilleur résultat est obtenu lorsqu'on enlève l'heuristique mra, l'algorithme résout alors le problème en 80 938 coups et 297 millisecondes. Il peut paraître étonnant de voir que l'algorithme ne résout plus le problème si on lui enlève l'heuristique $ab1$ dont le rôle est d'accélerer la détection du gain. L'explication est que pour $n = 2$ l'*alpha-beta* principal n'étudiera jamais plus que deux coups pour couleurordi.

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x	x	13467738	42454
$sans\ tt$		x	x	x	x	x	x	»	»
$sans\ tri$	x		x	x	x	x	x	»	»
$sans\ ab1$	x	x		x	x	x	x	»	»
$sans\ ab2$	x	x	x		x	x	x	»	»
$sans\ cqt$	x	x	x	x		x	x	4055427	12579
$sans\ mra$	x	x	x	x	x		x	3423233	10766

Tableau 7.4. *Cross-Cut 7x7 bas avec* $n = 5$

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x	x	133722	422
$sans\ tt$		x	x	x	x	x	x	159701	500
$sans\ tri$	x		x	x	x	x	x	109966	375
$sans\ ab1$	x	x		x	x	x	x	non	non
$sans\ ab2$	x	x	x		x	x	x	non	non
$sans\ cqt$	x	x	x	x		x	x	118686	390
$sans\ mra$	x	x	x	x	x		x	80938	297

Tableau 7.5. *Cross-Cut 7x7 bas avec* $n = 2$

Si on considère uniquement cet *alpha-beta*, cela veut dire que si il trouve une solution gagnante finale elle devra toujours être étudiée dans les deux premiers coups. Or l'*alpha-beta* secondaire n'a pas cette limite. Il etudie tous les coups correspondant à sa profondeur (par exemple pour $x = 5$ il etudie toutes les mises à deux degrés de liberté). Il peut donc trouver des solutions que ne trouve pas l'*alpha-beta* principal.

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x	x	359372705	1215968
$sans\ cqt$	x	x	x	x		x	x	33491	1063
$sans\ mra$	x	x	x	x	x		x	23902801	73453

Tableau 7.6. *Cross-Cut 7x7 haut avec* $n = 5$

Le tableau 7.6 montre que pour le Cross-Cut 7x7 haut et $n = 5$ les heuristiques cqt et mra sont aussi néfastes. La meilleure résolution est obtenue sans cqt.

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ ms$
$toutes$	x	x	x	x	x	x	x	30066	110
$sans\ tt$		x	x	x	x	x	x	33268	110
$sans\ tri$	x		x	x	x	x	x	non	non
$sans\ ab1$	x	x		x	x	x	x	»	»
$sans\ ab2$	x	x	x		x	x	x	52334	172
$sans\ cqt$	x	x	x	x		x	x	32540	110
$sans\ mra$	x	x	x	x	x		x	28635	109

Tableau 7.7. *Cross-Cut 7x7 haut avec* $n = 2$

Le tableau 7.7 donne les résultats pour le Cross-Cut 7x7 haut et la meilleure valeur trouvée pour n, soit $n = 2$. On voit que tri et $ab1$ sont nécessaires pour le résoudre rapidement. La meilleure résolution utilise 28 635 coups et 109 millisecondes.

$Heuristique$	tt	tri	$ab1$	$ab2$	cqt	mra	$ncoups$	$Xcoups$	$temps\ en\ h$
$toutes$	x	x	x	x	x	x	x	»	>10h
$sans\ cqt$	x	x	x	x		x	x	1354240513	1h 14m
$sans\ mra$	x	x	x	x	x		x	»	>10h

Tableau 7.8. *Goban 6x6 vide avec* $n = 9$

Le tableau 7.8 montre que sans les coups qui tuent, Dariush résout le goban 6x6 vide avec 1 354 240 513 coups en 1 heure et 14 minutes.

De façon surprenante, l'heuristique des coups qui tuent donne de mauvais résultats la plupart du temps alors que pour d'autres jeux, y compris d'autres sous jeux du jeu de Go comme la vie et la mort des groupes, elle donne habituellement de bons résultats. Les heuristiques cqt et mra semblent efficaces sur des *alpha-beta*s de petites profondeurs, moyennement efficaces sur des profondeurs moyennes et très pénalisantes sur des alpha betas de grande profondeur.

7.5. Conclusion

Nous avons présenté des heuristiques de tri et de sélection des coups appliquées à la résolution d'Atarigo. Ces heuristiques ont permis de résoudre Atarigo 6x6 pour un goban vide et pour un goban 7x7 avec un Cross-Cut. Il est vraisemblable que ces heuristiques peuvent être utilisées dans d'autres jeux. Nos travaux futurs consisteront à tenter de résoudre Atarigo 8x8 avec un Cross-Cut et à tester les heuristiques sur d'autres problèmes.

7.6. Bibliographie

[BOI 06] BOISSAC F., MARCHAND E., Les 3 logiciels Dariush, http ://ricoh51.free.fr/, 2006.

[CAZ 02a] CAZENAVE T., « Admissible Moves in Two-Player Games », *SARA 2002*, vol. 2371 de *Lecture Notes in Computer Science*, Kananaskis, Alberta, Canada, Springer, p. 52-63, 2002.

[CAZ 02b] CAZENAVE T., « Gradual Abstract Proof Search », *ICGA Journal*, vol. 25, n° 1, p. 3-15, 2002.

[CAZ 02c] CAZENAVE T., « La Recherche Abstraite Graduelle de Preuves », *Proceedings of RFIA-02*, Angers, France, p. 615-623, 2002.

[CAZ 03] CAZENAVE T., « A Generalized Threats Search Algorithm », *Computers and Games 2002*, vol. 2883 de *Lecture Notes in Computer Science*, Edmonton, Canada, Springer, p. 75-87, 2003.

[CAZ 04] CAZENAVE T., « Generalized Widening », *ECAI 2004*, Valencia, Spain, IOS Press, p. 156-160, 2004.

[MAR 86] MARSLAND T., « A review of game-tree pruning », *ICCA Journal*, vol. 9, n° 1, 1986.

[WER 02] VAN DER WERF E., UITERWIJK J., VAN DEN HERIK H., « Solving Ponnuki-Go on Small Boards », UITERWIJK J., Ed., *The 7th Computer Olympiad Computer-Games Workshop Proceedings*, Maastricht, The Netherlands, IKAT, Department of Computer Science, Universiteit Maastricht, p. 5-11, 2002.

[WER 05] VAN DER WERF E., AI techniques for the game of Go, Phd thesis, Universiteit Maastricht, Maastricht, The Netherlands, January 2005.

[WOL 00] WOLF T., « Forward pruning and other heuristic search techniques in tsume go », *Information Sciences*, vol. 122, p. 59-76, 2000.