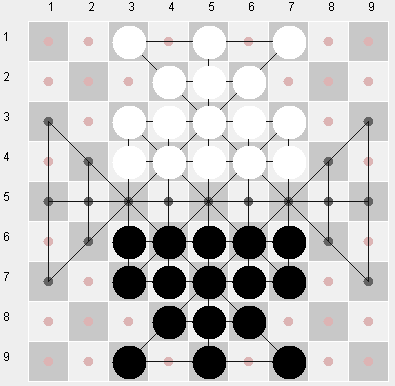
Intelligence Artificielle : 16 Soldats



Sommaire

[Introduction – 16 Soldats 3](#_Toc359779810)

[Présentation du jeu 3](#_Toc359779811)

[Règles du jeu retenues 3](#_Toc359779812)

[Enjeu du projet 4](#_Toc359779813)

[Réalisation du projet 5](#_Toc359779814)

[Mise en œuvre technique 5](#_Toc359779815)

[Mise en œuvre algorithmique 6](#_Toc359779816)

[Difficultées rencontrées 6](#_Toc359779817)

[La légalité des mouvements 6](#_Toc359779818)

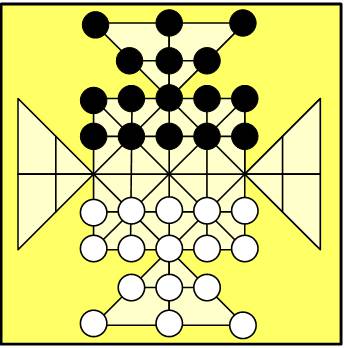
[L’utilisation des ressources matérielles 9](#_Toc359779819)

[Points à améliorer 10](#_Toc359779820)

# Introduction – 16 Soldats

## Présentation du jeu

16 Soldats est un jeu de stratégie sur plateau, à deux joueurs, originaire du Sri Lanka. Il serait aussi partiellement originaire d’Inde, où le jeu est connu sous le nom de « Vaches et Léopards ». *16 Soldats* est similaire à l’Alquerque : les joueurs peuvent utiliser leurs pièces (les soldats) pour sauter par-dessus les pièces adverses et les capturer. Cependant, le jeu se déroule sur une version étendue du plateau de jeu de l’Alquerque : quatre plateaux triangulaires se greffent au plateau principal (carré). Les pièces peuvent de surcroit capturer dans toutes les directions. Chaque joueur dispose de 16 soldats, dotés d’une couleur unique (par joueur).

  
Plateau de jeu « 16 Soldats »

## Règles du jeu retenues

Nous n’avons pas retenu la totalité des règles de 16 soldats pour la réalisation de ce projet. Seules les règles suivantes ont été intégrées :

* Le jeu se joue sur un damier de 9x9 cases, en suivant les lignes de déplacement autorisés (dessinées sur le plateau de jeu).
* La position initiales des pièces est toujours identique (dessinée sur le plateau de jeu).
* Le joueur utilisant la couleur Blanche commence.
* Une pièce est capturée lorsque un pion saute par dessus, mais toujours en suivant la direction initiale du déplacement, en poursuivant la ligne dans la même direction. Le déplacement est donc en ligne droite, de deux segments de la même ligne. On ne peut capturer qu’une seule pièce par tour. Il n’y a pas de prises multiples (comme par exemple on peut en trouver aux dames). Si un pion saute au dessus d’un pion ennemi, ce dernier est obligatoirement capturé.
* La capture de pièce n’est pas obligatoire quand elle est possible.
* Si l’un des deux joueurs ne peut plus jouer, la partie est déclarée nulle.
* La partie est limitée à une durée de 10 minutes.
* La partie est limitée à 200 coups (au total).
* Les colonnes et lignes du plateau de jeu sont numérotées de 1 à 9, (1,1) étant le point supérieur gauche de celui-ci.

## Enjeu du projet

Le but du projet est de développer, en binôme, une intelligence artificielle capable de jouer une partie de 16 Soldats. L’enjeu étant évidemment de développer la meilleure IA possible, afin de remporter le *grand tournoi*. L’IA doit respecter les règles suivantes :

* Ne pas écrire sur le disque dur (lecture autorisée).
* Ne pas effectuer de calculs pendant le tour de l’adversaire.
* Respecter les règles du jeu.

Pour ce faire, nous avons utilisé les notions étudiées en cours, telles que le parcours d’arbre des coups possibles (avec un nombre défini de « coups d’avance »), ainsi que le calcul d’heuristique.

# Réalisation du projet

## Mise en œuvre technique

|  |  |
| --- | --- |
| http://www.digitaltrends.com/wp-content/uploads/2010/11/java-logo.jpg  Java 7 | Conformément aux exigences du projet, nous avons utilisé le langage **Java** (version 1.7) pour développer notre intelligence artificielle. Le code « squelette » imposé nous proposait une interface IJoueur qui décrivait les méthodes à définir pour développer un joueur virtuel. |

|  |  |
| --- | --- |
| Nous avons créé un dépôt Subversion sur Google Code pour y *versionner* toutes nos classes. Les principaux avantages d’un gestionnaire de version comme **Subversion** sont : faciliter le partage du code, du développement en équipe, de la une sauvegarde du projet à chacune de ses évolutions. | http://www.geckogeek.fr/wp-content/uploads/2009/09/subversion_logo-300x259.png  Subversion |

|  |  |
| --- | --- |
| http://www.nosyweb.fr/images/stories/notepad_logo.png  Notepad++ | L’IA a été développée à l’aide de **Notepad++**, un éditeur de code multi-langage célèbre. En effet, le squelette de projet fourni ne semblait pas adaptable dans un environnement de développement intégré (IDE) sans faire, au préalable, certaines modifications. Nous avons préféré directement utiliser un éditeur de code pour gagner du temps de développement. Le *debug* a donc été effectué, tout au long du projet, avec l’outil « **jdb** » : Java Debugger. |

## Mise en œuvre algorithmique

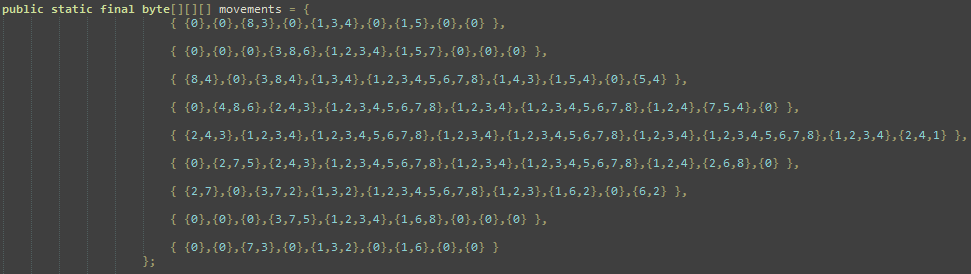
Avec notre solution, le joueur peut prévoir jusqu’à 7 coups d’avance de façon relativement rapide. Cette configuration dépend évidemment de la puissance de la machine hôte. Nos tests, réalisés sur des ordinateurs portables (respectivement un Intel core i5 avec 4go de mémoire vive et un Intel core i7 avec 8go de mémoire vive) permettent de monter jusqu’à 9 « générations » (9 coups d’avance) sans problème de gel ou de réflexions excédant la minute. À chaque coup (une configuration plateau de jeu) est associée une valeur (obtenue via une heuristique) qui nous permet de déterminer sa qualité (est-ce un bon ou un mauvais coup ?).

Nous avons choisi d’utiliser l’élagage alpha-beta, qui est une amélioration de l’algorithme minimax. L’heuristique associée à chaque nœud de notre arbre des coups possibles correspond à la différence de soldats amis et de soldats ennemis. Le but étant de maximiser le nombre de soldats amis à chaque coup (soit en capturer un maximum et en perdre un minimum).

## Difficultées rencontrées

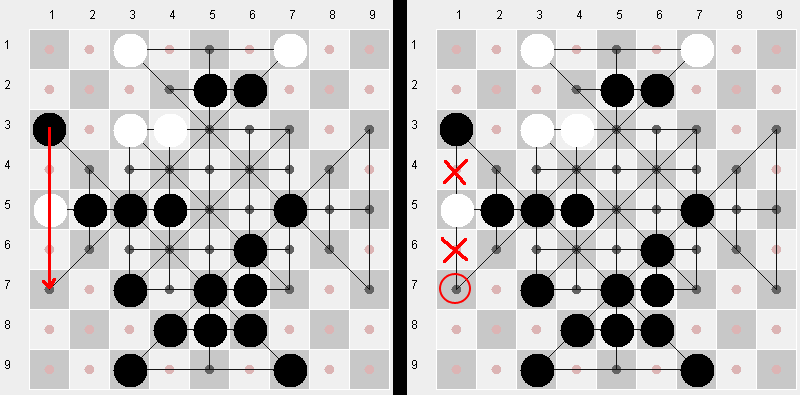
### La légalité des mouvements

Les règles du jeu font que la légalité de certains déplacements des soldats est parfois complexe à établir. C’est notamment le cas des mouvements présentés ci-dessous, qui ont été résolus avec l’écriture de quelques contrôles ; contrôles mis en œuvre à l’aide du tableau à 3 dimensions suivant, décrivant les directions autorisées à partir de chaque case du plateau de jeu :

Tableau 3 dimensions des directions autorisées

#### La capture de pion dans les plateaux triangulaires

Les plateaux triangulaires ont la particularité d’occuper 5 cases du plateau avec leur bord le plus long, pour seulement 3 cases « traversables ». Les soldats ont donc besoin d’avancer de 2 cases pour bouger d’une seule, ou de 4 s’il y a un soldat ennemi à capturer sur le chemin.

**  
Déplacement « doublé » sur les bords des triangles

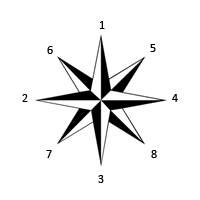
La nécessité de sauter une case supplémentaire (ou deux) est identifiée lors du parcours de chaque direction autorisée par le tableau à trois dimensions :

|  |
| --- |
| nextCol **=** **(**BestSoldier**.**movements**[**i **+** nextCol**][**j **+** nextLine**].**length **==** 1 **?** BestSoldier**.**colMov**[**BestSoldier**.**movements**[**i**][**j**][**k**]-**1**]\***2 **:** nextCol**);**  nextLine **=** **(**BestSoldier**.**movements**[**i **+** nextCol**][**j **+** nextLine**].**length **==** 1 **?** BestSoldier**.**rowMov**[**BestSoldier**.**movements**[**i**][**j**][**k**]-**1**]\***2 **:** nextLine**);** |

Si la case visée (qui est autorisée d’après le tableau 3D) n’autorise aucune direction (elle n’en contient qu’une : « 0 »), alors on sait que l’on doit doubler le déplacement occasionné par cette direction (movements[i][j][k] \* 2). Les déplacements sont stockés en mémoire dans les tableaux colMov et rowMov :

|  |
| --- |
| **// Attention : L'index démarre à 0. colMov[0] correspond au déplacement en colonne du mouvement "1"**  **public** **static** **byte[]** colMov **=** **{** 0**,** **-**1**,** 0**,** 1**,** **+**1**,** **-**1**,** **-**1**,** **+**1**};**  **public** **static** **byte[]** rowMov **=** **{** **-**1**,** 0**,** 1**,** 0**,** **-**1**,** **-**1**,** **+**1**,** **+**1**};** |

Pour rappel, les directions numérotées suivent l’ordre établi par cette rose des vents :

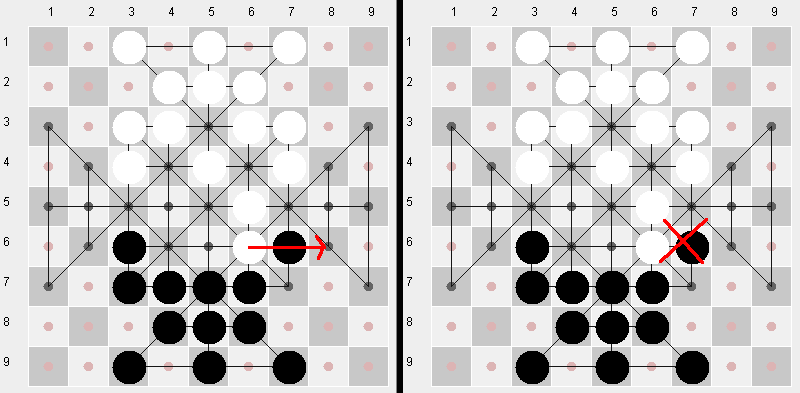


Numérotation des directions dans le programme

Ainsi, colMov[0] rowMov[0] correspond à la direction « 1 », soit : ne pas changer de colonne et remonter d’une ligne. Ce qui explique le déplacement (0, -1).

#### Ne capturer qu’en ligne droite

Une erreur basique consiste à penser qu’on peut capturer un pion s’il est à coté d’un pion allié et que la case qui le suit est « traversable » (=> elle contient plusieurs directions possibles). Cependant, il faut penser à vérifier que la case en question est atteignable **depuis la case où se trouve l’ennemi**.



Exemple de déplacement interdit à prendre en compte

Nous avons donc insérer une vérification qui nous permet de ne pas tenir compte du mouvement (la capture du pion adverse) si jamais la case qui le suit n’est pas atteignable en respectant les règles du jeu.

### L’utilisation des ressources matérielles

Notre première version du programme construisait un arbre de tous les coups possibles, plateau de jeu par plateau de jeu, d’une profondeur allant de 3 à 5, afin de prévoir 1 à 2 coups amis. La quantité de plateaux de jeu possible entraînait une occupation mémoire colossale. Notre programme allouait ainsi jusqu’à 1600Mo de mémoire vive et entraînait une interruption du programme Java. De plus, le parcours de l’arbre lors de la décision du meilleur coup consommait une grande partie du temps de calcul alloué par le processeur. En effet, plusieurs fois, les 4 cœurs de notre machine de test (Intel core i5) ont été sollicités à 100%. Une partie avec une profondeur d’arbres de 5 était alors parfois impossible à finir.

Nous avons premièrement réalisé une importante économie de mémoire en utilisant des données de type «byte » (et non « int » que nous utilisons généralement). En effet, chaque nœud utilisait une matrice de type int[9][9], entrainant une grande consommation de mémoire (81x4 octets par nœud). Avec des variables de type byte, notre programme ne consommait plus « que » 800 à 1000Mo de mémoire vive, mais le processeur était toujours autant sollicité.

Suite à cette première amélioration, après une tentative d’implémentation d’un tableau de hachage (sans amélioration notable), nous avons décidé de ne conserver qu’un seul plateau de jeu en mémoire lors du calcul des heuristiques. Les avantages de cette méthode sont nombreux :

* Une économie de mémoire vive très significative
* Nous n’effectuons plus de *deep-copy* des matrices byte[9][9] à chaque nœud mais modifions seulement le plateau de jeu courant
* Nous calculons l’heuristique de chaque nœud lors de sa création (ainsi que celle de ses nœuds-fils)
* Nous ne créons plus de nouveau nœud si nous élaguons avec alpha-beta

Les améliorations sont ainsi flagrantes : le programme ne consomme plus qu’environ 70mo de mémoire vive et le processeur n’est jamais sollicité à plus de 20% (sauf pic de charge). Il est aussi bien plus rapide puisque nous pouvons utiliser des arbres de profondeurs allant de **7** à **9** sans aucun problème, avec fluidité.

## Points à améliorer

Bien que notre programme s’exécute assez rapidement en calculant 7 à 9 coups d’avance et en consommant peu de mémoire vive, certains points peuvent être améliorer pour proposer au joueur adverse un plus grand challenge, notamment :

* L’heuristique : Assez simple, le calcul de la différence des soldats amis et ennemis donne un premier aperçu de la qualité du coup à jouer, mais reste sommaire. Nous pourrions calculer une heuristique plus complexe et plus complète en utilisant les possibilités de déplacement, ou l’anticipation de technique de capture particulière (piéger l’adversaire, etc).
* L’amélioration de l’heuristique permettrait de créer des parties intéressantes en faisant s’affronter deux joueurs utilisant notre IA. Actuellement, les cas où celles-ci finissent dans une boucle infinie stoppée par la limite de 200 coups sont encore fréquents, **malgré l’ajout d’un tirage aléatoire des prochains coups possibles** (toujours avec pour objectif d’obtenir la meilleure heuristique).